THE UNIVERSITY OF ALBERTA

PARTITIONING AND ASSIGNMENT OF PROGRAMS FOR DISTRIBUTED SYSTEMS

by

© ATUL GUPTA

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF Master of Science

Department of Computing Science

EDMONTON, ALBERTA

FALL 1982

dedicated to my parents

ABSTRACT

Distributed systems are becoming increasingly popular and feasible due to falling cost of hardware brought about by LSI and VLSI technology, and advances in computer interconnection networks. The primary reason for this popularity is the benefits of extensibility, reliability, and performance promised by distributed systems that are difficult (if not imposible) to achieve by uniprocessors, multiprocessors, or computer networks. But before these benefits can be realised, certain problems peculiar to distributed systems must be solved.

Partitioning refers to dividing up a program into a number of tasks such that constraints in the original program are satisfied and parallelism between tasks is maximized. Past approaches to the partitioning problem are reviewed and their limitations outlined. A new approach is suggested that can extract more parallelism out of the program than previous approaches.

The next step after the partitioning of a program is the assignment (allocation) of these tasks to processors. A new heuristic algorithm is presented that takes precedence relationships between tasks into account. The complexity of the algorithm is seen to be linear in the number of tasks and inter-task communication paths. Experimental results indicate that this algorithm is fast and produces good assignments. This algorithm is applied to calculate the bounds on the number of processors needed to execute a set

of partially ordered tasks in the least time.

A salient feature of distributed systems is that the communication between tasks takes place to explicit message passing on the communication network. To facilitate this type of communication, the system provides a set of communication primitives. These primitives should satisfy certain requirements so that the benefits promised by these systems can be materialised with the greatest of ease. These requirements are presented and argued to be essential.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Dr. Subrata Dasgupta, for his constant encouragement throughout the course of this work. He always asked the right questions at the right time. He constantly encouraged me to sharpen my writing skills.

I am indebted to Rajender Razdan for his careful reading of the earlier drafts of this thesis. Also, I am grateful to Dr. Rick Heuft, Dr. Francis Chin and Sundar Iyengar for their comments on this work.

Finally special thanks to my parents, to whom this work is dedicated.

Table of Contents

List of Figures

Chapter 1

Introduction

Distributed systems are becoming increasingly popular and feasible due to falling cost of hardware brought about by LSI and VLSI (Very Large Scale Integration) technology, and advances in computer interconnection networks. Another reason for this popularity is the benefits (i.e. growth, reliability, protection) promised by distributed systems that are difficult (if not impossible) to achieve by centralized systems.

1.1 What is a Distributed System?

There is considerable confusion among researchers in Computer Science as to the meaning of the term *distributed system*; or what are the characteristics of such systems. Thus it is essential for us to define it precisely so as not to leave any room for ambiguity in the later discussion.

We subscribe to the definition suggested by Jensen [JENS78]. He defines a distributed system as a

> "...multiplicity of processors that are physically and logically interconnected to form a single system, in which overall executive control is exercised through the cooperation of decentralized system elements. Conceptually, a single executive manages all of the system's physical and logical resources in an integrated fashion, but its kernel logic (perhaps hardware as well as software) and data structures are replicated among a number of processors and memories. These executive kernel copies are individual entities that execute concurrently, asynchronously, synergistically, and without hierarchical master/slave relationships, to form a single organism."

It should be emphasized at this point that distributed processing can exist in three different dimensions – architecture, operating system, and application programs [ECKH78]. A system not exhibiting distribution in all the three dimensions cannot be classified as a "distributed system". An example would be the Cm* system [SWAN77] that exhibits distribution in architecture and application programs but lacks a decentralized operating system, and hence cannot be called a "true" distributed system.

The user views a distributed system as a single coherent entity and its "existence is totally transparent ...[ENSL78]." So the user initiates an action and specifies what is to be done and not how this action is to be accomplished by the cooperation of individual system elements. Thus all the details of the system structure and operations are hidden from the user.

Another important characteristic of distributed systems is that there is no centralized control; all the elements in the system are highly autonomous in nature. There is no master/slave relationship; all the elements in the system make decisions in line with the overall system objective.

1.2 Advantages of Distributed Systems

Certain advantages accrue naturally from distributed systems. First, distributed systems are more *extensible* than centralized, multiprocessor, and network computers.

Extensibility refers to the

> "...degree to which system functionality and performance can be changed without changing the system design [JENS78]."

Distributed systems can be modified easily in the sense that a hardware component can be replaced without affecting other parts of the system. Moreover, performance and functionality of the system can be extended in small steps upto a fixed upper limit.

Next, they provide better system reliability. Since the services and computations are not localized at one place, the system can withstand mechanical, algorithmic and subsystem failures. In addition, these systems are fail-soft as the physical and logical coupling is not strong among system elements.

Thirdly, distributed systems are capable of providing greater performance. A major overhead in uniprocessor systems is context-switching (because it requires saving registers, memory, files, protection information, timing, accounting etc.) when more than one job is competing for processor cycles. In a distributed system, a job will immediately activate an idle processor and hence provide a smaller response time. This will result in processor inefficiency (as the processors are not multiprogrammed) but it is tolerable due to cheap processing power offered by the system.

1.3 Problems in Designing Distributed Systems

A major benefit can be derived from distributed systems if a number of processors work concurrently and synergistically on a single job. This would necessitate partitioning the job into tasks and assigning these tasks to individual processors.

As noted by Jensen [JENS78], job (program) partitioning techniques are still very primitive in nature. Partitioning can be either performed by the programmer or the compiler. The programmer can partition the program on the basis of computing capabilities of system elements, data accessing patterns, logical relationships between program parts, maximizing parallelism etc. But this approach is not promising for it would be too much to expect from the programmer. Many different approaches have been tried to get the compiler to do the partitioning. As the current state in partitioning now stands, it would be difficult to be sure which particular approach is the best without further experience gained from an actual compiler implementation.

It is unlikely that an efficient algorithm would be found for assigning tasks to processors as this problem is conjectured to be NP-complete. Consequently, efficient heuristic algorithms need to be developed to perform task assignment.

The tasks assigned to processors communicate via messages. The first stage in design of any inter-task

communication mechanism would be to analyze the current and future requirements of such communication. Only then can various forms of inter-task communication mechanism can be designed. These are only some of the major problems associated with designing distributed systems.

1.4 Plan of the Thesis

The next two chapters of this thesis present the problem of program partitioning and assignment for distributed systems, and the current approaches to the solution of the problem.

Chapter 4 describe a new heuristic algorithm for the program assignment problem.

Chapter 5 discusses analytical measures for bounding the number of processors needed to execute a given set of partially ordered tasks, and then a new approach is suggested.

Chapter 6 presents issues involved in designing inter-task communication mechanism.

Finally, chapter 7 presents conclusions and suggests further areas for research.

Chapter 2

Program Partitioning Problem

The widespread use of distributed systems is hampered by the lack of effective methodologies for designing programs to be run on such systems. There are two approaches to programming an algorithm for such systems – programming in a sequential or in a parallel programming language (e.g. CSP [HOAR78], DP [HANS78]).

The first approach requires partitioning of a program into tasks and then assigning these tasks to processors in the system. *Partitioning* refers to the division of a program into tasks such that inter-task communication cost is minimized, parallelism between tasks is maximized, and precedence constraints in original program are preserved. A sequential program will not exhibit any parallelism. Thus the system component performing partitioning must uncover all the parallelism in the program within a reasonable time limit and constraints.

Assignment is the process of allocating tasks to available processors such that it meets certain objectives and/or constraints (e.g. minimization of inter-processor communication cost, minimization of completion time). It is assumed that task execution and inter-task communication cost is measured in units of time. If need arises, cost measured in units of time can be readily converted to dollars. It is conjectured [ULLM73] that the problem of optimal assignment of tasks is NP-complete. Thus the

objective would be to look for efficient heuristics which produce results that are close to optimal. Partitioning can be either performed by the compiler or software designer and assignment will be a key component of the operating system.

The problems encountered in the second approach are not apparent. It is possible to structure a program as a set of concurrent processes. But this structuring of programs totally ignores costs incurred owing to communication among processes that could limit applicability of this approach in distributed computing environment. This suggests that a communication oriented approach will give a different view of the problem and may lead to a better understanding of the distributed program design methodology.

In the following sections we review the work done in the area of partitioning of programs as applicable to distributed systems. We also point out the weaknesses and strengths of each of these approaches. Improvements are also suggested to enhance their generality and improve performance.

2.1 Partitioning of programs

This section is devoted to a discussion of partitioning techniques. The partitioning problem can be formulated as follows : given a serially coded program, recognize the set of tasks and their precedence relationship such that the program can be executed by a given distributed

system in compliance with certain objectives. These objectives would be minimization of execution time when the tasks do not use more than a given number of message transfers or minimization of overall program execution time. It should be emphasized that we are not banking on the programmer to specify the parallelism in the program, but the job of detecting parallelism is relegated to the compiler and supervisory programs.

One of the first attempts at designing algorithms for detecting parallelism was made by Fisher [FISH67]. His algorithm is based on the conditions for parallel processing developed by Bernstein [BERN66]. These conditions can be expressed as follows. Two tasks T1 and T2 can be executed in parallel if the following conditions are satisfied.

$$I1 \cap O2 = \emptyset$$
$$I2 \cap O1 = \emptyset$$
$$O1 \cap O2 = \emptyset$$

where I1 and O1 are the input- and output sets of the task T1. In particular, input set of a task refers to the set of variables required to execute the task.

The algorithm takes a FORTRAN program as its input and divides it into tasks. These tasks could be single or a set of statements. These tasks are, then, analyzed and input/output sets calculated. Once the input and output sets of every task are known, it is straightforward to generate [BERN66] essential ordering relations among tasks.

The tasks that can be executed concurrently are easy to determine from this essential ordering. A major drawback of this approach is its inability to detect intra-statement parallelism. That is, it does not take advantage of the parallelism in arithmetic statements, not to mention the DO-loops or recurrence relations. Moreover, it does not consider execution and inter-task communication time.

Recently, Allan and Oldehoeft [ALLA79] used this technique to translate high level language programs to data flow programs. We suggest here that the original program can be translated into a data flow program and then this data flow program can be partitioned into a set of partially ordered tasks. Since data flow graphs [DENN80] represent operator dependencies and token flow naturally, the job of estimating execution and communication time is much simplified. Furthermore, these data flow graphs exhibit parallelism at a much more microscopic level which would not be possible [ACKE82] by conventional method of calculating input/output sets for every single statement. Thus the partition of the program can be optimized with respect to the communication and execution cost (henceforth, we will use the term communication cost and communication time interchangeably).

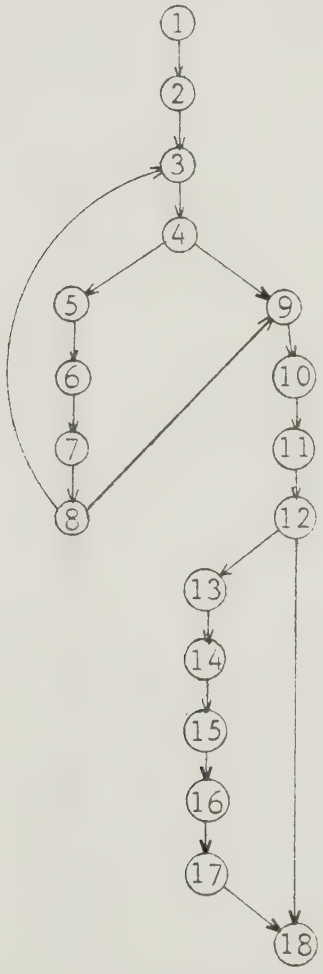
Ramamoorthy and Gonzalez [RAMA69] presented another approach to detection of task parallelism. The FORTRAN *parallel task recognizer*, developed by them, takes a FORTRAN program as input and numbers each of the executable

statements starting from the first executable statement. It also identifies the *program graph* which is the sequence of tasks executed in the original program. The program graph is represented as a connectivity matrix C where Cij is 1 if and only if there is an arc from node i (task i) to node j (task j). Figure 2.1(a) gives an example of a FORTRAN program, the corresponding program graph is shown in Figure 2.1(b), and the connectivity matrix of the program in Figure 2.1(c). The recognizer replaces each of the maximally connected subgraphs by a single node and alters the program graph and connectivity matrix accordingly. Iterative constructs (like Do-loops) give rise to maximally connected subgraphs and must be transformed to a single node. After this transformation, the program graph will not contain any loops or strongly connected components. The program graph after removal of maximally connected components is shown in Figure 2.1(d).

The next step is to derive the final program graph and connectivity matrix T. Each of the tasks in the reduced program graph are examined for their inputs and outputs to build up the matrix T. An element Tij of T is 1 if and only if task j uses one of the outputs of task i. Using the connectivity matrix T and the conditions developed by Bernstein [BERN66]; it is easy to form precedence partitions. The program graph of parallel processable program and precedence partitions are shown in Figure 2.1(e). Each of the members of precedence partition

```
1.          READ 100, (A1(I),I=1,10), B, C, D
2.          READ 100, (A2(I),I=1,10), NS, NST, NSTU
3.          DO 10 I = 1, 10
4.          IF (A1(I)-A2(I)) 20, 20, 40
5.    20    X1 = A1(I)*(B-C)
6.          X2 = A2(I)+(B/C)
7.          A3(I) = X1*X2
8.    10    CONTINUE
9.    40    CALL ALPHA(A1, A2, ABC, B4, B5)
10.         PRINT 3057, X1, X2, (A3(I), I=1,10)
11.         CALL BETA(X1, X2, A3, B6)
12.         IF (B4-B5) 50, 50, 60
13.   50    READ 325, E, F, G, H
14.         X3 = (E*F)+(G-H)
15.         X4 = B6+G
16.         X5 = X3-X4
17.         X6 = (B4+B5)*X5
18.   60    PRINT 4, X3, X4, X5
```

(a)

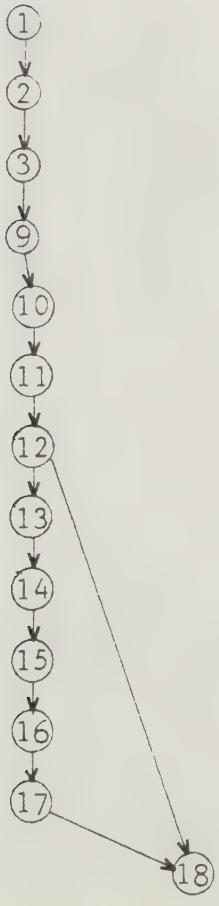


(b)

Figure 2.1 (a) A FORTRAN program and
(b) its program graph

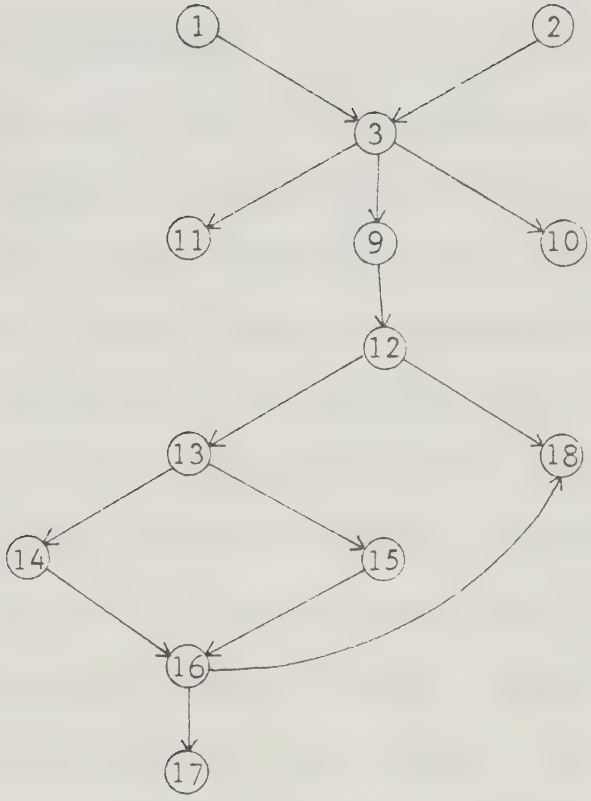
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
8	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1
13	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(c)



(d)

Figure 2.1 (c) connectivity matrix of the program graph
(d) the reduced program graph



(e)

Precedence partitions {1,2}, {3}, {9,10,11}, {12}, {13}, {14,15}, {16}, {17,18}

Figure 2.1 (e) Program graph of parallel processable program

P_i can be executed in parallel after the tasks in the previous partition P_{i-1} have been completed.

The FORTRAN parallel task recognizer only serves to identify the tasks that can be executed in parallel. It makes no effort to reduce the total execution time by considering task execution time, inter-task communication time and loop optimizations. However, it spawned research efforts in this area. One of them was the design of a compiler for DYNAMO.

DYNAMO is a well known simulation language that is characterized by simplicity of the syntax, data structure and relative independence in the order of execution of statements. These features prompted Huen et al [HUEN77] to design a compiler that produces code for the TECHNEC, a unidirectional ring network of 12 LSI-11 nodes. Message communication is the bottleneck for parallel computations on TECHNEC. Since every processor has only 12K words of RAM, the compiler is implemented as a pipeline. Each stage of the pipeline refers to a different part of the compiler namely scanner, macro expansion, symbol table routines, parser, sequencing module, code generator and partitioning module. The code for each stage of the pipeline resides at a different processor. All the stages of the pipeline execute in parallel and communicate with each other by passing messages.

An important part of the compiler is the partitioning module which produces tasks that are to be executed in

parallel. The main objective of the partitioning module is to produce partitions that will reside on different computers and run concurrently in order that program runs in minimum time. The input to the module consists of storage and execution time requirement of each of the statements along with a list of variables used in the program. The module uses this information to derive a precedence graph and subsequently partitions. The partitioning problem is posed as Mixed Integer Linear Programming form and then standard mathematical programming techniques are applied. The main drawback of the design is that the partitioning algorithm used is highly dependent on the features of DYNAMO.

El-Dessouki et al. [ELDE79] described techniques to implement a partitioning compiler for an ALGOL-like language. The algorithms described by them fall into two categories :

1. partitioning, and
2. assignment.

Since a discussion of assignment techniques is not the intent of this chapter, we will point out the salient features of the partitioning algorithm.

Tesler and Enea [TESL68] have suggested the concept of a *single assignment language* which, in simple terms means that a variable may appear on the left hand side only once in a program. The programs written in a single assignment notation have the property [ACKE82] that the parallelism

inherent in the program can be detected in a straightforward manner. The partitioning module of [ELDE79] transforms the program into a single assignment program and treats each statement as a single task. The next step is to define a dependency relation among tasks using Bernstein's [BERN66] conditions. The algorithm also computes communication cost, storage requirements and execution cost of each of the tasks.

The partitioning module can handle references to arrays and simple variables, but no suggestion is offered as to support of other types of data structures *viz* linked list, tree, stack etc. The research is still continuing in this area and only an implementation can give an indication of the practicality of an approach.

The previous two paragraphs discussed techniques to translate a high-level language program to a form suitable for execution on a distributed system by transforming it to a single assignment language program. One may, then, wonder – why not write the program in single assignment language itself? The answer to this question can be found in a recent proposal by Christopher et al [CHRI81]. They have designed a single assignment language – SALAD – for distributed applications. Since programs written in SALAD are already in single assignment language form, the partitioning techniques suggested in [ELDE79] can be used here also.

Kuck and coworkers [KUCK79] have been working on the design of a compiler for high speed multiprocessors. Their model of a multiprocessor consists of processor clusters (containing p processors each) interconnected via alignment network to the global memory. A salient feature of the structure is the control of a processor by a set of status bits that can be manipulated by other processors in the system. Since the status bit of a processor can be manipulated by other processors, they are not completely independent. Nevertheless, these processors may be executing different instructions in the program and hence, the system can be classified as an MIMD (Multiple Instruction Multiple Data Stream) [FLYN66] type computer. The techniques they discuss for transforming an iterative construct into a maximal parallel form are applicable to our model. Specifically, their algorithms fall in three categories :

1. partition,
2. algorithm change, and
3. loop freezing.

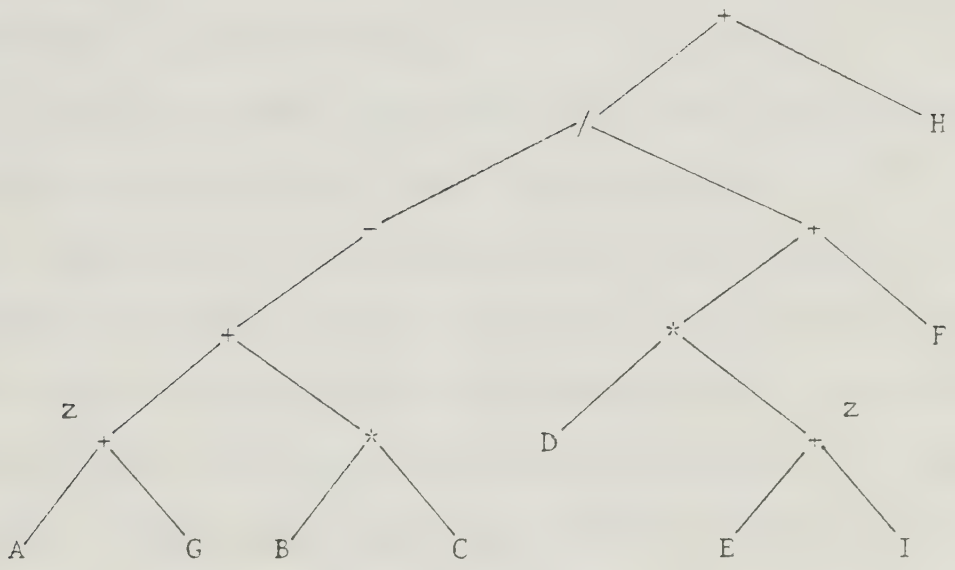
These algorithms are derived from corresponding algorithms for array processors. Thus these will not be described here (the interested reader may refer to [KUCK79]). A noteworthy feature of the proposal is the prominence accorded to other parts of the system (*viz* interconnection network, synchronization mechanisms, data transmission) in the overall context of the partitioning process. These other

issues will probably play as important a role in the partitioning process as the partitioning algorithm itself. Thus future research work in this area should consider issues like the type of interconnection network, inter-task communication, synchronization mechanisms etc.

It has been found [KUCK76] that a major portion of parallelism in ordinary high-level language programs is in arithmetic and iterative statements. It is imperative that efficient algorithms and representations of programs will be developed to extract parallelism in arithmetic statements and iterative constructs. A scheme developed by Wang and Liu [WANG79] is in the same spirit. They have investigated the problems involved in parallel execution of arithmetic expressions in high-level language programs. An arithmetic expression, represented as a tree, can be compiled into a *Parallel Execution String* (PES) [WANG79]. PES is defined as a path from a leaf to the root of the tree. Thus each PES will be a sequence of constants, variables and operators. An arithmetic expression is shown in Figure 2.2(a) and its corresponding tree in Figure 2.2(b). The Parallel Execution Strings generated are shown in Figure 2.2(c). For a detailed description of the algorithm and notation, the reader may refer to [WANG79]. A salient feature of the PES is that two operators on two different paths (PES) can be executed concurrently before their merging point. Referring to Figure 2.2(b), operators marked by "z" can be executed concurrently because they are on different PES and below

```
-(A+G+B*C)/(D*(E+I)+F)+H
```

(a)



(b)

```
A + G + #1 - / #2 + H
B * C + #1 - / #2 + H
E + I * D + F /' #2 + H
```

(c)

Figure 2.2 (a) The arithmetic expression,
(b) its parse tree, and
(c) the corresponding PES.

their merging point (the root).

The PES concept can be extended from an arithmetic expression to the complete program [WANG79]. The idea presented is innovative in the sense that we are able to extract more parallelism from arithmetic statements. But no mention is made as to how to handle other type of statements *viz* IF-THEN-ELSE, iterative constructs, procedure calls. More work is required in these areas before it can be applied to real-world situations. In the passing we note that each of the PES can be considered as a task to be assigned to a processor.

The next step, after partitioning of the program, is to assign these tasks to available processors. It is possible that the number of processors are more than the number of tasks. In that case the tasks will not wait for the availability of the the processors and so in theory it is possible to achieve the minimum finish time. But the problem of optimal assignment of tasks on more than three processors has been conjectured to be NP-complete. In the next chapter, we discuss the possibilities in assigning tasks to processors. Whenever there is a choice between brevity and clarity in our presentation, we will look for the latter. Wherever possible, concepts will be explained by examples.

Chapter 3

The Task Assignment Problem

The previous chapter discussed the program partitioning problem in an informal manner. Partitioning produces a set of partially ordered tasks along with their estimated execution time. The partitioning procedure should provide, if possible, an estimate of inter-task communication time. Thus, the program may be represented by a precedence graph in which each vertex represents a task and precedence relationships are designated by directed edges. Each of the vertices are assigned a weight equal to the execution time of the task and weights assigned to edges represent inter-task communication time. Moreover, it is also assumed that loops are eliminated by the partitioning procedure [RAMA69].

Given two or more processors, the problem is to decide which processor should be executing a given task from the set of tasks at any instant of time. This assignment must not violate any of the precedence relationships or the requirement that not more than one processor is assigned to a task at a time. It is also assumed that once a processor starts executing a task, it must work uninterrupted on this task until it is finished. The reason for this assumption is the belief that task preemption is a burden on system resources like memory, data channels, CPU cycles etc. Henceforth, the discussion will be restricted to non-preemptive task assignment.

In a distributed system of homogeneous processing elements it may be possible to assign different tasks of a program freely among different processors. But the interconnection network may become an inherent bottleneck in the system.

One may hope that throughput will increase in same proportion as the number of processors. Thus, it is expected that throughput will double by doubling the number of processors to execute a program which has parallelism greater than number of available processors. But experience with distributed systems has shown that throughput follows the "actual" rather than "ideal" curve as shown in Figure 3.1. Chu [CHUW78] and Jenny [JENN77] give examples of this phenomenon. This behaviour has been attributed to excessive interprocessor communication which depends on the number of interconnected processors. Since the overhead of interprocessor communication reduces the number of machine cycles available for useful computation (as some processing of messages is required), this job could be delegated to an autonomous module. This approach is followed by Arnold and coworkers [ARNO82] in designing Modular Missile Borne Computer (MMBC) for missile and space borne defence applications.

In the systems where high costs are imposed for interprocessor communication, processors are assigned tasks in such a way as to minimize interprocessor communication. However, if the processors in the system are interconnected

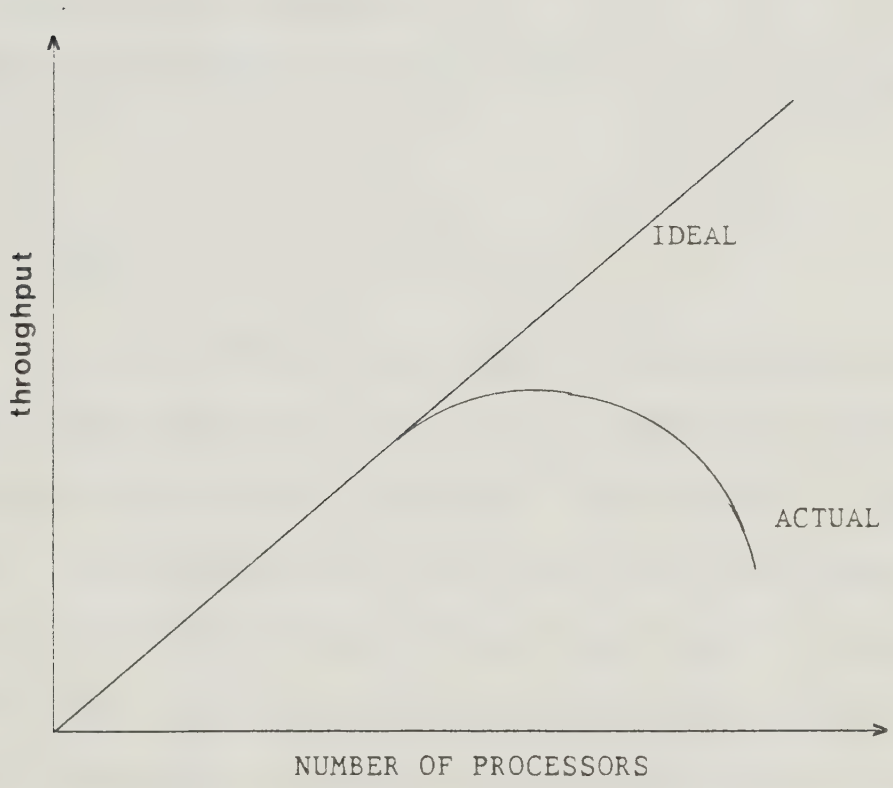


Figure 3.1 The saturation effect

by high bandwidth data paths, as in the CRYSTAL system being developed at University of Wisconsin-Madison [LAND82], many feasible assignments of computational tasks to processors would exist. In these cases, interprocessor communication would no longer be considered as a bottleneck and so minimum completion time of the program should be the objective.

The different proposed approaches to the assignment problem fall into one of the three categories :

1. graph theoretic,
2. integer programming, or
3. heuristic methods.

These approaches are discussed in following sections.

3.1 Graph theoretic approach

Stone [STON77] and other researchers (then at the Brown University) introduced a graph theoretic approach to the task assignment problem. This work is mostly concerned with a multiprocessor model derived from computer graphics application where tasks may float from a powerful central processor to a terminal oriented satellite processor. Note that these two processors may execute concurrently on different programs, but never on the same program. Or in other words, execution of a program shifts dynamically from one processor to another. That is, only one processor would be executing some part of a program at any instant of time.

In this method, tasks are represented as vertices and edges as inter-task references. Weights are attached to edges representing communication cost. An inter-task communication (ITC) cost of zero means no communication takes place between two tasks and an ITC cost of infinity means that these two vertices must be assigned to the same processor.

A unique source vertex, corresponding to the processor P1, and a unique sink vertex corresponding to the processor P2, act as source and sink for the maximum network flow algorithm employed. The execution costs of tasks are represented by adding two edges to each of the task vertices in the graph. The cost of running a task on P1 is denoted by the weight of the edge joining that task vertex to vertex P2 and vice versa.

The objective of task assignment in this method is to minimize total cost which is defined as the sum of processing cost and inter-processor communication (IPC) cost. Obviously, these costs are dependent on the task-to-processor assignment.

Each cutset of this graph divides the vertices into two disjoint subsets, with processor vertices P1 and P2 in different subsets. All the task vertices in the subset containing P1 are assigned to processor P1. Similarly, the rest of the task vertices are assigned to processor P2. It is not difficult to see that this graph is a commodity flow network [HORO78]. With this direct correspondence between

cutsets and task assignment, it is obvious that a minimum weight cutset of the graph will yield minimum cost task assignment.

Consider, for example, the modified graph [STON77] shown in Figure 3.2(b). As explained earlier, the cost of running task A on processor P1, which is 5 units, is indicated in the graph by an arc joining vertex A and P2 with a weight of 5 units. The minimum weight cutset algorithm on the graph produces the cut-set indicated by the double line. This is the minimum cost assignment of given tasks among two processors.

In order to solve an n-processor problem, Stone [STON77] suggests that it may be reduced to several two-processor flow problems :

> "...we show that a two-processor flow can give information about the minimal cut in an n-processor graph, but we are unable to produce a complete efficient algorithm."

Stone conjectures that n^2 two-processor flows need to be solved to find a solution to the n-processor problem. Nevertheless, it is conjectured [ULLM73] that for more than two processors the problem is NP-complete.

An algorithm suggested by Wu and Liu [WUSB80] produces an approximate n-cut by applying the minimum weight cutset algorithm in the order of n times. Let us assume that the original graph is modified by adding n processor nodes, then the n-cut is the subset of edges that partitions the modified graph into n disjoint subsets each containing one

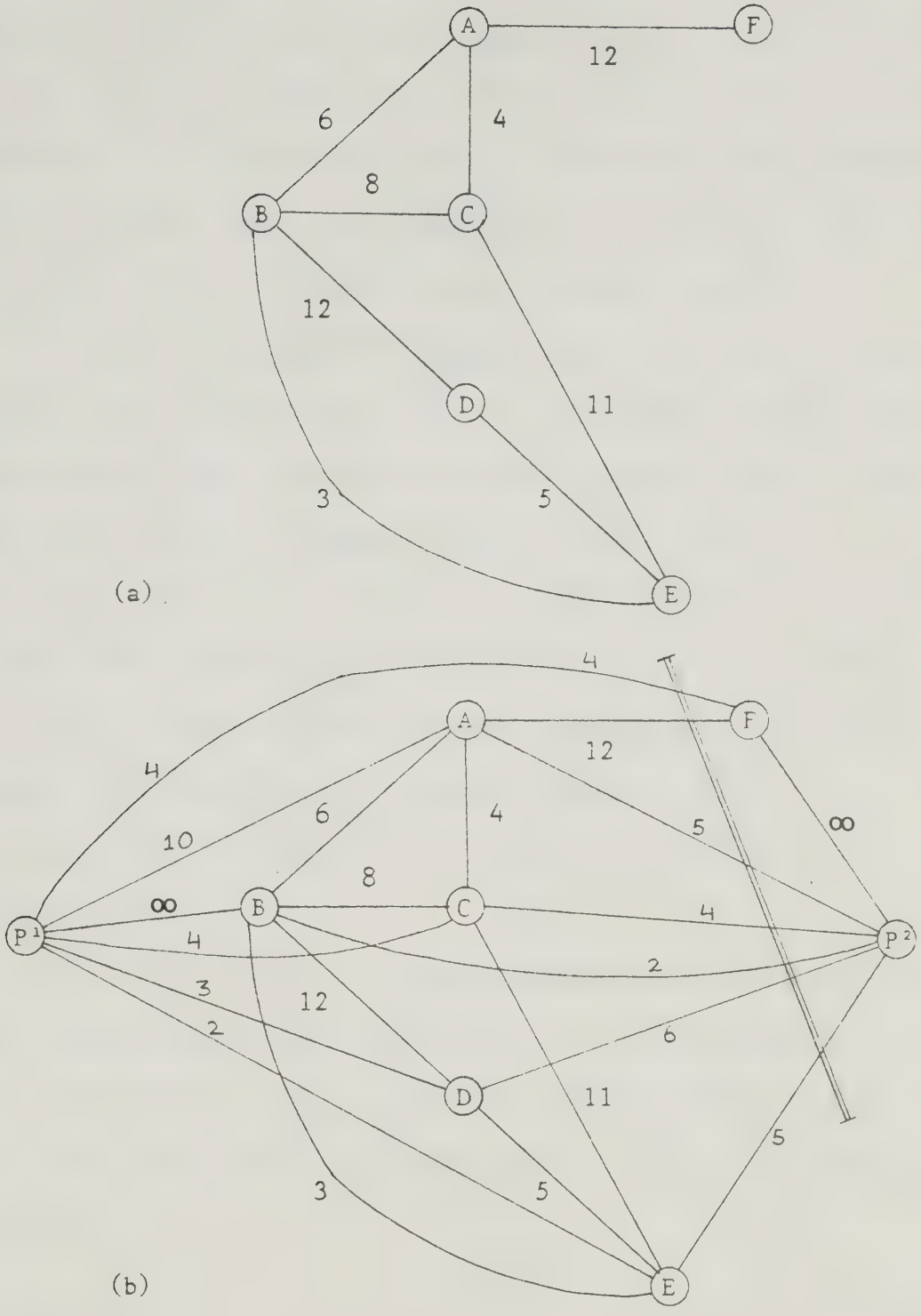


Figure 3.2 (a) Inter-task communication graph, and
(b) the minimum communication cost cut.

and only one processor node. Computational results presented for small problems (7 tasks and five processors) indicate that the solution generated is close to optimal, but the results cannot be extrapolated to larger problems. Moreover, it is possible that precedence relationships exist between tasks. But no mechanism is available to indicate these precedence relationships between tasks.

Lo and Liu [LOVI81] suggested a group of efficient heuristics to find the n-cut of the task graph. The three algorithms they suggested can be classified as iterative, lump, and greedy, respectively. Their simulation results with 20 tasks and 5 processors indicate that the latest finish time produced by heuristics is within 10% of the solution produced by an optimal algorithm in 94% of the cases. The heuristics produced good results but, as before, incorporating precedence constraints in the model limits its usefulness.

There have been attempts [RAOG79] to include memory size constraints but the solution is NP-complete. Thus it can be inferred that graph theoretic considerations may not give the most efficient solution to the task assignment problem.

3.2 Integer programming

The integer programming approach, as applied to the task assignment problem, is basically derived from optimization techniques. In this approach, the task assignment problem is formulated as an optimization problem and the technique of mathematical programming is then applied to solve it.

Let the cost of communication between tasks Ti and Tk be denoted by Cik. Also, let Eij represent the cost of running task i on processor j. This accounts for the fact that a task may have different execution costs on different processors (i.e. the system is heterogeneous). In order to represent the assignment of tasks to processors, define an element Xij such that it is 1 if task Ti is assigned to processor Pj; otherwise it is 0. The constraint that any task is assigned to only one processor can be represented as :

$$\sum_{j=1}^{n} Xij = 1 \qquad \forall\ i$$

Other constraints like limited memory, load balancing, etc can be incorporated into the model depending on the application environment.

The assignment problem can be formulated [PRIC81] as a 0-1 quadratic programming problem as follows :

$$\text{Cost}(X)=\sum_{i=1}^{m}\sum_{j=1}^{n} Eij*Xij+\sum_{i=1}^{m}\sum_{k=i+1}^{m} Cik-\sum_{i=1}^{m}\sum_{j=1}^{n}\sum_{k=i+1}^{m} Cik*Xij*Xkj$$

where m and n are number of tasks and processors, respectively. The first summation term represents the execution cost for each task on its assigned processor. The second term is the total communication cost among tasks. The third term accounts for the fact that two tasks which communicate with each other during program execution incur no inter-processor communication cost if assigned to the same processor. Thus these terms are subtracted from the total cost.

Price [PRIC81] used an iterative algorithm which starts with an initial assignment and repeatedly reassigns tasks to processors until further improvement is not possible. This reassignment is performed by transforming the assignment matrix X (i.e. set/reset an element Xij of the matrix X). The basic idea is to calculate the "penalty" matrix P, where Pij is defined as the cost of executing task i on processor j for all values of i and j and then select the most profitable reassignment. The complexity of this algorithm is $O(n*m^3)$. Our implementation of this algorithm expended more than 40 CPU-seconds to find a solution for 40 tasks and

7 processors. Moreover, the final assignment is dependent on the initial assignment of tasks. Experimental results indicate that this algorithm is time consuming and does not produce good results. Furthermore, the assumption that the tasks float from one processor to another is not realistic.

Recently Ma et. al. [MAPR82] applied 0-1 programming technique to solve the task assignment problem. This technique can be thought of as branch-and-bound (BB) method as defined by Kohler and Steiglitz [KOHL76]. The task assignment problem is represented as a decision tree. There are m levels (one for each task) in the tree and each internal node has n branches (corresponding to the number of processors). Each of these branches are labeled by processor names from P1 to Pn. A branching decision at each internal node is equivalent to assigning the task at that level to the processor attached to that branch. These branching decisions are subject to the constraints discussed earlier. Thus, a path from the root to the last node is a complete assignment.

An example [MAPR82] of a three level decision tree with three processors and a feasible path (indicated by double lines) is shown in Figure 3.3. At level one the branch traveled corresponds to processor 3 and so task 1 is assigned to processor 3. Similarly tasks 2, and 3 are assigned to processors 1, and 2 respectively.

The algorithm generates an optimal solution if the objective is to minimize the inter-processor communication

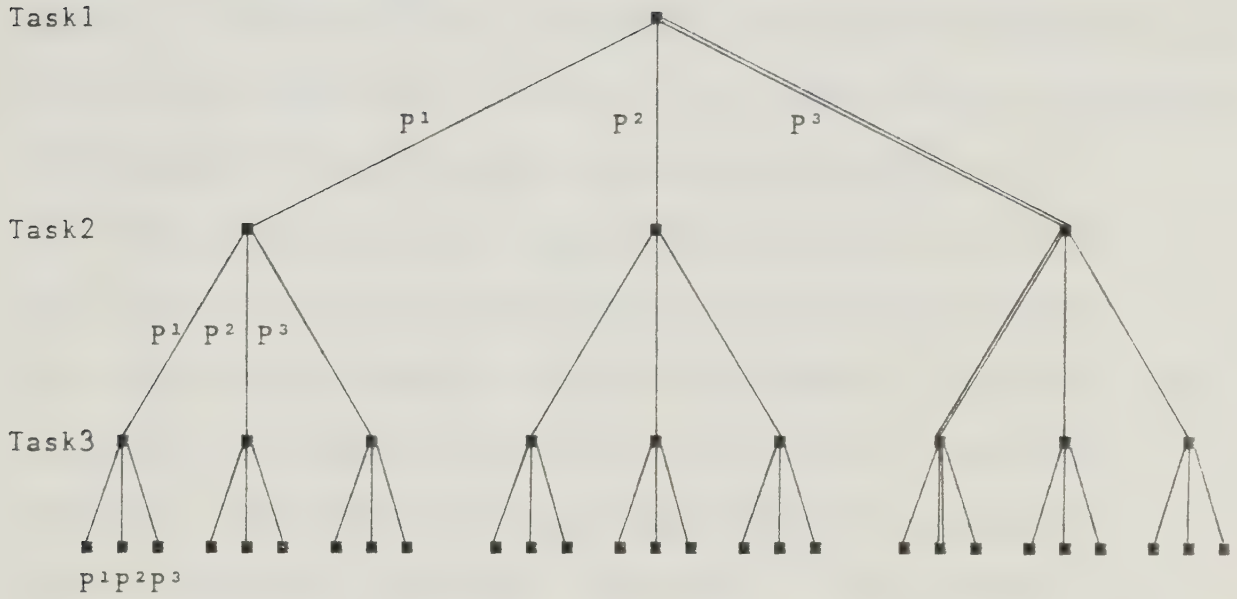


Figure 3.3 Decision tree with three tasks three processors

cost. Moreover, it can incorporate various engineering requirements *viz* small memory size, processor size constraint, task redundancy. The algorithm is computationally inefficient and hence cannot be applied to time critical applications. But it can be used in cases where only one program runs on the system (i.e. the program is permanently tied to the system) and so a near-optimal assignment is a must.

Goal programming [IGNI76] can be applied to the assignment problem to produce an approximate solution. Recently, Ignizio et al [IGNI82] suggested an augmented goal programming approach to solve large scale design problems in distributed computing and supersystem design.

The augmented goal programming algorithm can be logically divided into two phases. The first phase of the algorithm calculates the initial assignment of tasks to processors by means of a fuzzy clustering algorithm [MCCO72]. The fuzzy clustering algorithm groups tasks in a cluster if these tasks are "similar". One *similarity* measure is high inter-task communication cost; thus two tasks with high inter-task communication cost will be assigned to the same cluster. The second phase accepts the output of the first phase as its input and tries to improve the assignment via an exchange search algorithm. Details can be found in [IGNI82].

Even though it is possible to incorporate constraints in the integer 0-1 programming method, it is still deficient

in some areas. Firstly, it is difficult to incorporate precedence relationships among tasks in the model. Since most of the application programs give rise to precedence relationship among tasks, this approach cannot be used. Secondly, the amount of CPU-time and memory required is prohibitively large. Thus it can be inferred that this technique will find limited usefulness unless methods are developed to overcome previous two shortcomings [CHUW80].

3.3 Heuristic methods

At this point it should be emphasized that finding an optimal solution is strictly an academic issue since the task assignment model itself is an approximation of real-world situations. Incorporating extra constraints or conditions in the model will only make the task of finding an optimal solution more difficult. In this situation, it would be advantageous to design algorithms that are efficient and produce good assignments. The following paragraphs discuss some heuristic algorithms for solving the task assignment problem.

Gylys and Edwards [GYLY76] proposed two algorithms for task assignment with the objective of minimizing IPC cost. The objective of minimization of IPC cost is inspired by several facts about the interconnection structure : it is unreliable, insecure and a potential bottleneck in the system.

The first algorithm forms task clusters with minimum inter-cluster communication cost. To form such clusters, a task pair with maximum inter-task communication cost – among all eligible pairs – is selected and checked if it satisfies certain constraints (e.g. memory loading, processor time). If these constraints are satisfied, the pair is fused into a single task and removed from the list of eligible pairs; otherwise it is removed from the list of eligible pairs and the process repeated. The algorithm stops when all the eligible pairs are exhausted. It is quite possible that the number of clusters identified are more than the number of processors available.

The second algorithm removes this deficiency. Briefly, the algorithm assigns an initial centroid for each of the clusters (equal in number to the number of processors). It then calculates the "distance" [GYLY76] of each task to the centroid of all the clusters, assigns the task Ta to cluster Ce such that Ta is nearest to Ce for all such pairs, and adjusts the centroid of cluster Ce. These steps are repeated until no task clusters change or the number of iterations exceed a predetermined limit.

Some of the disadvantages of this algorithm are : long search times are required before it can find an eligible task; no consideration is given to precedence relationships among tasks; and no attempt is made to balance loads among the processors.

Efe [EFEK82] has suggested an improved algorithm, based on Gylys and Edwards's work, that takes load balancing into account. Efe's algorithm can be divided into two phases. The first phase identifies all the task clusters to minimize IPC cost (and is similar to the second algorithm suggested by Gylys and Edwards [GYLY76]). The second phase of the algorithm checks the output of the first phase for any violation of the load balancing constraint. In that case, some of the tasks from overloaded processors are shifted to underloaded processors by a task reassignment algorithm (TRA). The TRA will stop only when a balanced load distribution is achieved. This algorithm, like the previous one, fails to take precedence relationships into account.

Arora and Rana [AROR80] suggested two heuristic algorithms for task assignment. Their distributed program model represents tasks and inter-task communication as nodes and edges of a graph, respectively. This graph is augmented with n nodes – corresponding to each processor – and edges between each processor-task node pair (as in Stone's algorithm [STON77]). Thus the task assignment problem is equivalent to finding an n-cutset of the augmented graph such that each subset contains one and only one processor node.

The first algorithm selects an arbitrary task node and coalesces that node with that processor or task node which has the edge having maximum cost among all edges incident upon the selected task node. The algorithm terminates when

all the task nodes are fused into processor nodes.

The second algorithm starts with an initial assignment and repeatedly reassigns tasks to achieve a better assignment. The algorithm terminates when no reassignment takes place in an iteration. As is true with most of the other algorithms, it is difficult to incorporate precedence relationships into this model.

Some attempts to design heuristic algorithms for assigning a set of partially ordered tasks – but with no ITC cost – have been reported in literature [KOHL75], [ADAM74]. Kohler [KOHL75] has shown experimentally that critical path scheduling is near optimal in most of the cases. The critical path length for task Ti is defined as the length of a longest path from Ti to the terminal node. The basic idea behind critical path scheduling is as follows : task Ti is executed before task Tj if the critical path of Ti exceeds that of Tj. Adam et. al. [ADAM74] discuss a family of schedules called List Schedules and show that highest level first discipline produces the best results.

As mentioned earlier, these algorithms ignore communication cost and this oversimplification precludes them from use in distributed systems where inter-task communication cost cannot be neglected.

The foregoing discussion provides the motivation for heuristic algorithms that take into account precedence relationships among tasks and produce a good assignment for time-critical applications. It should be pointed out that

distributed systems are being designed that contain a large (~100) number of processors [LAND82]. In the near future, it is possible that for most of the applications the number of available processors is greater than the number of tasks. Thus attention should be paid to the design of assignment algorithms that work for an unlimited number of processors.

In the next chapter, we discuss two algorithms for the assignment of tasks. To the best of our knowledge, this is the first attempt to design an heuristic algorithm that incorporates precedence relationships also. Examples are given to clarify some of the concepts presented.

Chapter 4

A New Algorithm

The discussion in the previous chapter motivates the design of a heuristic algorithm to assign the set of partially ordered tasks on a homogeneous distributed processing system. By "homogeneous distributed processing system", we mean a set of homogeneous processors communicating with each other via a fully interconnected network. Moreover, the interconnection network is assumed to be constructed from high bandwidth fibre-optic links. This is a pragmatic view rather than a simplifying assumption.

Since processors are connected by high bandwidth links, the interconnection network may not be a bottleneck in the system. Thus we can shift our goal from minimizing inter-processor communication cost to minimizing completion time where completion time is the finish time of the last task in the program. Here we assume that the execution and communication cost is measured in terms of units of time.

The programs to be executed on the system are assumed to have already been partitioned into tasks. Each of these tasks contain a set of instructions which are to be executed sequentially on a processor and directed arcs between the tasks specify the execution order. Thus a task can send results to more than one successor tasks which could be executed on many (different) available processors. Some earlier approaches assume that execution of a program shifts

from one processor to another [STON77], [PRIC81] or only one processor is executing the program at any given instant of time. It goes without saying that this assumption is not realistic. Thus our approach is a radical departure from earlier approaches to this problem in the sense that it approximates real-world situation.

The basic strategy of the algorithm is as follows. Let us assume that we have a partial assignment of tasks and we have selected a task v to be next assigned to one of the processors. Among a range of alternatives, we select the best alternative such the completion time of v is the minimum possible.

4.1 Program model

We can represent the program by a directed acyclic graph (DAG) G = (V,E) with a finite set of vertices V and edges E such that each edge e has a head h(e) $\in$ V and a tail t(e) $\in$ V. We regard the edge e as leading from h(e) to t(e), or that edge e leaves h(e) and enters t(e). There is a unique source vertex s such that no edge enters s and a sink vertex *k* such that no edge leaves k. As before, vertices represent tasks and edges denote inter-task communication.

With each vertex v $\in$ V we associate a non-negative quantity vtx_cost(v) which is the estimated execution time of vertex v. Also, we associate a quantity edge_cost(e) for

each edge e $\in$ E to account for inter-task communication time.

4.2 The algorithm

Our minimum finish time algorithm can be logically divided into two phases. The first phase calculates the cost (in units of time) of the longest path from each and every vertex to the sink vertex. In short, this phase assigns priorities to all the vertices in the graph. The second phase uses the information gathered in the first phase to assign the tasks to processors. We now describe the first phase.

4.2.1 Phase 1

A path $p = e^1, e^2, \ldots, e^K$ is a sequence of edges such that $t(e^i) = h(e^{i+1})$ for $0 < i < k$ beginning with the vertex $h(e^1)$ and ending in $t(e^K)$. The cost of such a path is defined as

$$\sum_{i=1}^{k} [vtx\_cost(h(e^i)) + edge\_cost(e^i)] + vtx\_cost(t(e^K)).$$

We associate a quantity cost(v) with each vertex v $\in$ V such that it is the maximum cost path among all the paths from v to the sink vertex k.

We can use a modified breadth first search (BFS) algorithm [HORO78] on the graph G^r to calculate cost(v) for each v ∈ V. The reverse G^r of a graph G is the graph formed by replacing each edge e with an edge e' such that h(e') = t(e) and t(e') = h(e). The graph G^r is represented by an adjacency list [HORO76]. The queue Q contains unvisited vertices of the graph and the procedure addq(q) adds a vertex v to the end of the Q.

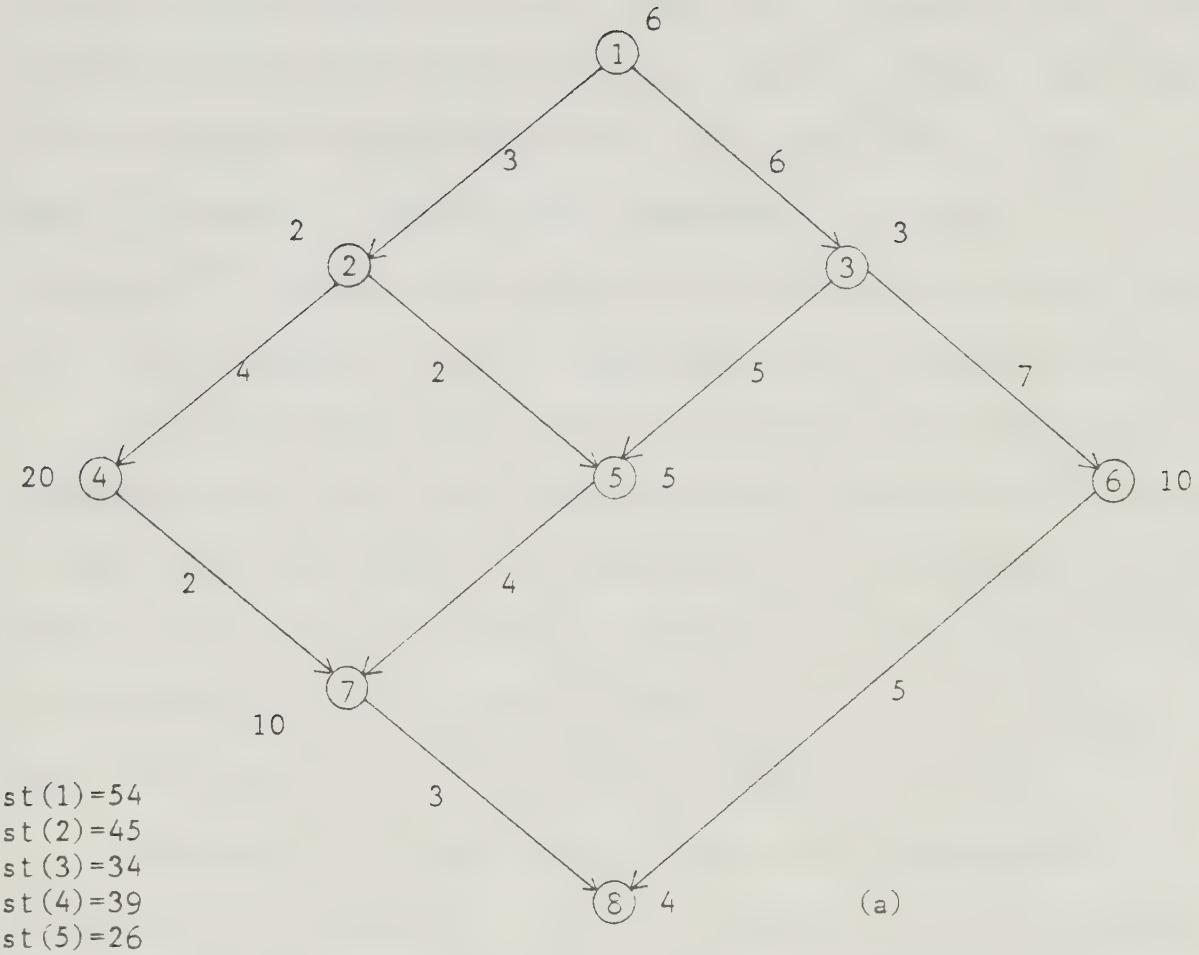
Initially, all the vertices are marked as unvisited and cost(v) is set to zero (by setting visited(v) to false and cost(v) to 0 for all v ∈ V). The sink vertex k of G will be the root vertex of G^r. Initially, only k is in the queue Q. A vertex u is deleted from the head of the queue and cost(w) calculated for all the vertices w adjacent to u. If these vertices have not been visited before, then they are added to the queue. After calculating cost(w) for all the vertices, visited(u) is set to false (indicating this vertex as visited). In the next iteration, a vertex is again deleted and the same operations are performed. The procedure stops when Q becomes empty.

The main advantage of the BFS is that it facilitates a quick look at the graph and identifies costly paths. the algorithm is expressed in an ALGOL-like language as follows:

```
procedure mbfs;
begin
   for all v ∈ V do cost(v):=0; visited(v):=false od;
```

```
   Q:=Ø;
   addq(k);
   while Q isnt empty do
      delete a vertex u from the head of Q;
      cost(u):=cost(u) + vtx_cost(u);
      for all w adjacent to u do
         let e ∈ E be the edge from u to w;
         if cost(w) < cost(u) + edge_cost(e) then
            cost(w):=cost(u) + edge_cost(e);
         if not visited(w) then addq(w) fi;
         visited(w):=true;
      od;
   od;
end mbfs;
```

Since each vertex is added to the queue only once and when a node u is deleted only the edges leaving node u are considered, the complexity of the algorithm is O(m + |E|) where |E| is cardinality of the set E and m is the number of vertices in the graph. Figure 4.1 illustrates the result of this algorithm. The "cost" array could also be calculated by a recursive algorithm.




```
cost(1)=54
cost(2)=45
cost(3)=34
cost(4)=39
cost(5)=26
cost(6)=19
cost(7)=17
cost(8)=04    (b)
```

Figure 4.1 (a) DAG and associated vertex costs
(b) output of the procedure "mbfs"

4.2.2 Phase 2

The second phase of the algorithm is intuitively quite simple. Assume that we have assigned vertex v to a processor and successors of v need to be assigned. There exists a vertex w such that

cost(w)+edge_cost(e(v,w)) > cost(u)+edge_cost(e(v,u))

for all successors u ∈ V of v and w≠u. Since w lies in the critical path, we should at least try to reduce the edge (communication) cost from v to w by assigning v and w to the same processor. Each of the successor vertices of v with indegree of one are assigned to different processors and the ones with indegree greater than one are assigned later.

Vertices with indegree greater than one are ready for assignment when all their predecessor vertices are assigned. In that case the algorithm makes one of two choices : either assigns this vertex v (say) to the processor which is also assigned one of v's predecessors or to a new processor so that the completion time of v is minimized.

The queue Q is the list of vertices ready to be assigned to a processor. The following procedure **assign** generates an assignment of tasks such that the finish time is minimal. We use an adjacency list [HORO76] to represent graph G.

```
procedure assign;
begin
   Q:=s;
```

```
pr_no:=1;
/*Assigns source vertex to processor 1*/
inset(s):=pr_no;
update vtx_finish(s), pr_finish(pr_no);
for all v ∈ V do count(v):=indegree of v od;
while Q isnt empty do
   delete a node u;
   if indegree of u > 1 then
      D:=[v|v is u's immediate predecessor];
      t1:=max[pr_finish(inset(t))],
         t ∈ D
               and let this vertex be w;
      t2:=max[vtx_finish(t)+edge cost from t to u];
         t ∈ D
      if t1 > t2 then
         /*Assign u to a new processor*/
         pr_no:=pr_no+1;
         inset(u):=pr_no;
         update vtx_finish(u),pr_finish(pr_no);
      else
         inset(u):=inset(w);
         update vtx_finish(u),pr_finish(inset(u));
      fi;
   fi;
   A:=[v|v is u's immediate successor];
   B:=[v|v ∈ A and indegree of v is 1];
   there exists w :
     cost(w)+edge_cost(e1) > cost(v)+edge_cost(e2),
```

```
        u,v ∈ B, h(e1)=u,t(e1)=w,h(e2)=u,t(E2)=v;
      /*Assign w to the same processor as u*/
      inset(w):=inset(u);
      for i from 1 to |B| do
         pr_no:=pr_no + 1;
         inset(v):=pr_no;   /*Assign v to a new processor*/
         update vtx_finish(v),pr_finish(inset(v));
      od;
      for all v ∈ A do
         count(v):=count(v) - 1;
         if count(v) = 0 then addq(v) fi;
      od;
   od;
end assign;
```

Some explanation is in order. To tell which processor vertex v belongs to we use inset(v). Arrays "pr_finish" and "vtx_finish" refer to the time a processor would finish executing tasks assigned to it and a task would finish executing on a processor, respectively. The processors are identified by "pr_no".

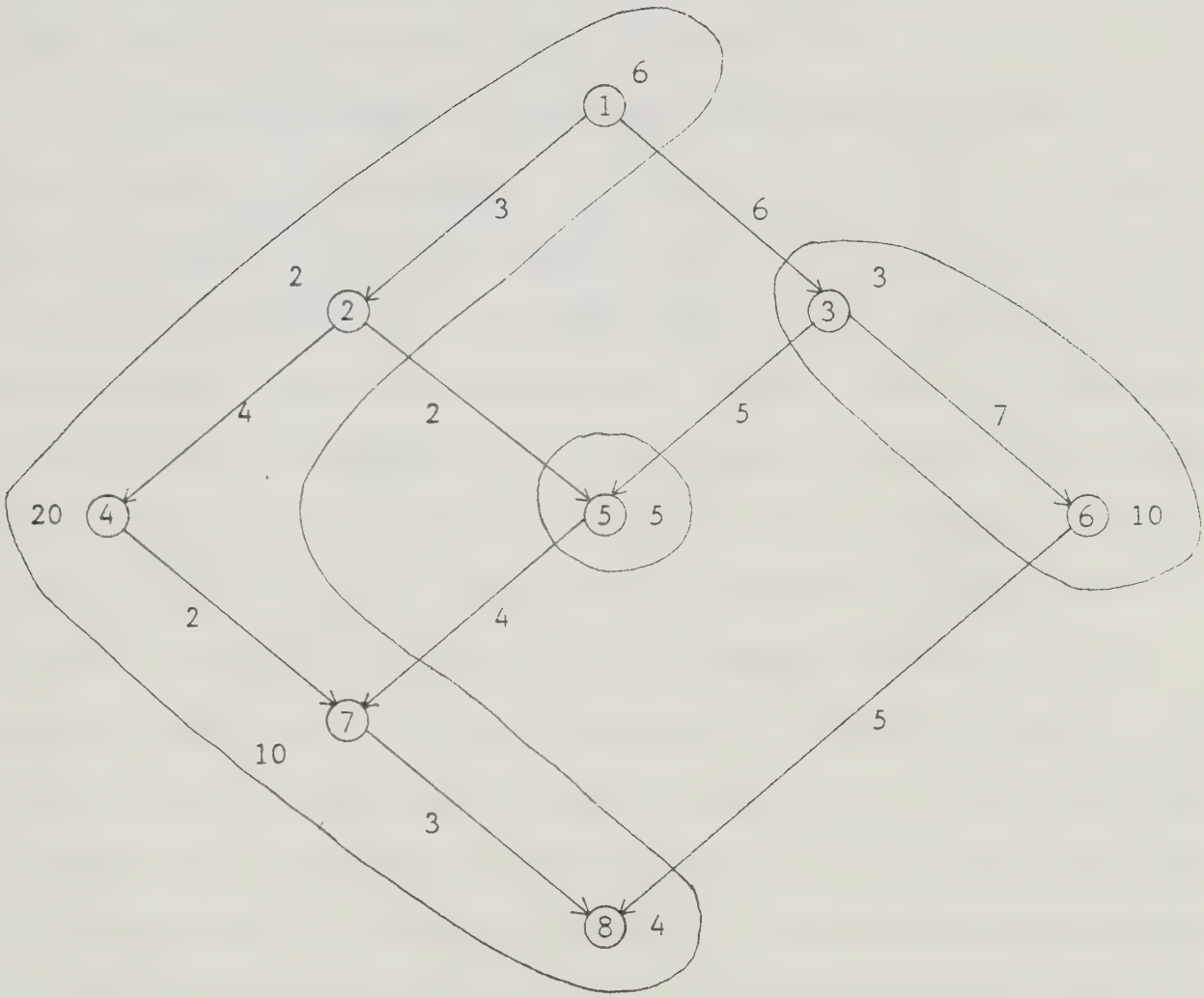
Briefly, the second phase of the algorithm works as follows. A vertex u is deleted from the queue Q and its indegree checked. If the indegree is greater than one, then all of u's predecessors have been assigned and the method described earlier is used. Otherwise the indegree is 0 or 1. In that case the procedure checks for all the successors of u and identifies a vertex w as part of the critical path.

Since w lies in the critical path, it is assigned to the same processor as u. The rest of the successors of u are assigned to different processors. The procedure returns when all the vertices have been assigned to processors. The final value of pr_no will indicate the number of processors used by the tasks.

Since each vertex is added to the queue only once and all the edges are considered atmost twice, the time complexity of this phase is O(m + |E|). Thus the overall complexity of the algorithm is O(m + |E|). Output of procedure **assign** for the graph of Figure 4.1 is given in Figure 4.2.

It should be evident by now that our algorithm and the method of critical path scheduling [KOHL75] resemble one another. But we argue that this resemblance is only superficial. Recall that a task communicating with another task residing on a different processor incurs communication cost. This communication cost is zero if these tasks reside on the same processor. As discussed in [KOHL75] the critical path length (cost) keeps on changing depending on the current state of task assignment. This should be evident when a task with an indegree of more than one is assigned.

So far we have discussed assigning tasks on an unlimited number of processors. But it is quite probable that the number of available processors is less than the number of processors calculated by the algorithm. In that



Total number of processors used = 3

Figure 4.2 Assignment of tasks produced by procedure "assign"

case we have to modify our algorithm. We briefly discuss the modification required in the next section.

4.3 Modified algorithm

The model of the program is same as before. This algorithm can be divided into two phases. The first phase is the same as the first phase of the previous algorithm. That is, it calculates the cost of the critical path from each vertex v $\in$ V to the sink vertex k $\in$ V.

The second phase of the algorithm assigns tasks to a given number of processors. To start with, all the tasks are arranged in non-increasing order of cost(v) for all v $\in$ V. Initially all the tasks that do not have any predecessor tasks are executable. These tasks are assigned to available processors. If the number of tasks is greater than number of available processors, then a *contention* for processors is said to take place. Whenever contention occurs, selection of tasks to be assigned immediately is made on the basis of "cost" values with higher "cost" valued task being assigned immediately. The selected task is then checked for its number of predecessors (i.e., the indegree of the vertex in the graph model). The appropriate method is applied (as discussed before) depending on the indegree of the vertex.

The assignment of tasks in Figure 4.1, for example, on two processors is shown in Figure 4.3. Notice that when

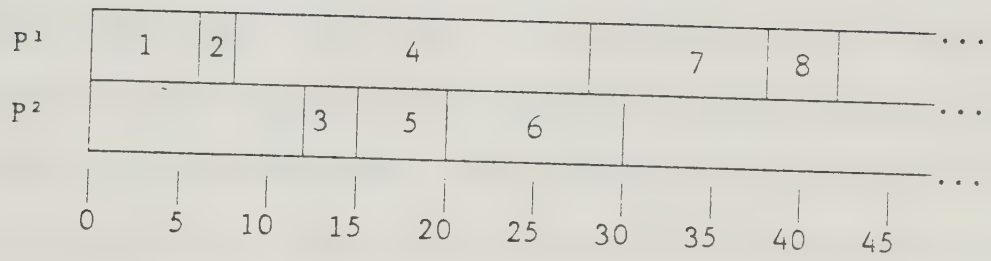


Figure 4.3 Assignment of tasks on two processors

task "1" completes execution on P1, tasks "2" and "3" are ready for assignment. Since task "2" has a higher priority, it is assigned to processor P1 and task "3" to P2. The task "3" cannot start execution before time unit 12 due to communication delay.

In the example we just discussed, all the tasks can finish execution by time unit 42 which, rather surprisingly, is same as the completion time of the assignment with unlimited number of processors (three processors to be precise). This suggests that it may be possible to achieve the same completion time using fewer processors by the modified algorithm. But it requires some knowledge of the approximate number of processors required to run this set of tasks in minimum time. Otherwise, if a small number of processors are specified, some of the tasks may remain idle due to lack of availability of processors.

This discussion motivates the calculation of a lower bound on the minimum number of processors needed to execute a given set of tasks in minimum time. This is an important aspect of task assignment and could prove to be beneficial in conjunction with our modified algorithm.

In the next chapter, we present a technique – based on the algorithms developed in this chapter – to calculate a lower and upper bound on the number of processors needed to execute a set of partially ordered tasks in minimum time.

Chapter 5

Bounding the Performance of Multiprocessors

The object of this chapter is to develop a lower and upper bound on the number of identical processors needed to execute a set of partially ordered tasks (taking communication delay into account) in the least time. Algorithms have been developed by several authors [CHEN68], [FERN73], [FERN75], [KRAS72] and [RAMA72] for finding these bounds with the assumption of no communication delay. These results may not be directly applicable to our model. In fact, we show by counterexamples – wherever possible – that methods proposed earlier when applied to our model may not give sharp lower bounds on number of processors.

The later part of the chapter introduces a new technique to derive bounds on the minimum and maximum number of processors. Computational requirements for calculating these bounds are also discussed.

5.1 Earlier Approaches

Chen and Epley [CHEN68] defined a very simple lower bound on the minimum number of processors. If $\sum t_i$ is the sum of all the task execution times and t_{CP} is the length of the critical path then their bound can be expressed as :

$$n = \left\lceil \frac{\sum t_i}{t_{CP}} \right\rceil$$

Note that the right hand side represents average processing activity per unit time to complete this set of computations in a time equal to the time of critical path of the graph. This lower bound does not contain a term reflecting partial ordering among tasks. Thus it will only provide a rough approximation to the true value. Moreover, it is difficult to incorporate communication time in this formula. A major advantage of this proposal is the the ease of calculating n. If m is the total number of vertices in the graph, then the time complexity of calculation of n is O(m).

Kraska [KRAS72] presented another lower bound for the number of processors. The *earliest completion time* of a task T_i is defined as the least time in which this task can be completed. Similarly, the *latest completion time* of a task T_j is the maximum time until which completion of this task can be postponed without increasing t_{CP} . If l is the maximum number of levels in the graph, then Kraska's bound can be expressed as

$$n = \left\lceil \max_{1 \le i \le l} \left\lceil \frac{P_i}{D_i} \right\rceil \right\rceil$$

where D_i is the largest of the latest completion times of the tasks at level i and P_i is the sum of the task times upto and including the tasks at level i.

Hu's [HUTC61] bound is defined for a graph with equal task times. This lower bound can be expressed as

$$n = \max_{1 \le i \le l} \left\lceil \left\lceil \frac{1}{i} \sum_{j=1}^{i} |L_j| \right\rceil \right\rceil$$

where $|L_j|$ is the cardinality of the precedence partition L_j. Basically, Hu's bound divides the total number of vertices executed by the time elapsed until a particular level is reached and takes the maximum over all the levels. The total number of vertices is counted by $\sum_{j=1}^{i} |L_j|$ and the level number corresponds to time elapsed since unit task times are assumed. Notice that Chen and Epley's bound is included in the Hu's bound for i=1. Moreover, Hu's bound contains effect of task distribution on the time axis.

Ramamoorthy [RAMA72] calculates the bound as follows. The task Ti are partitioned into *earliest precedence partitions* Ei such that tasks in Ei can be executed at the earliest time corresponding to level i. Another set of partitions called *latest precedence partitions* Lj are formed such that the tasks in Lj must be completed by the end of level j. Thus, the tasks that are common to partitions Ek and Lk have to be executed at time instant k and are identified as "essential tasks". Thus the maximum of essential vertices over all the levels is the lower bound on processors, or

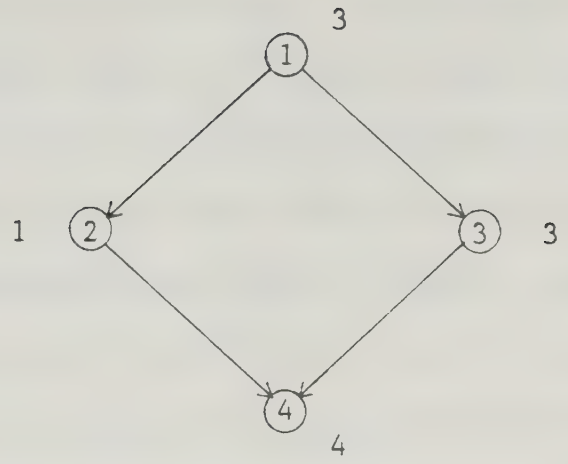
$$n = \max_{1 \le k \le l} \left\lceil |L_k \cap E_k| \right\rceil$$

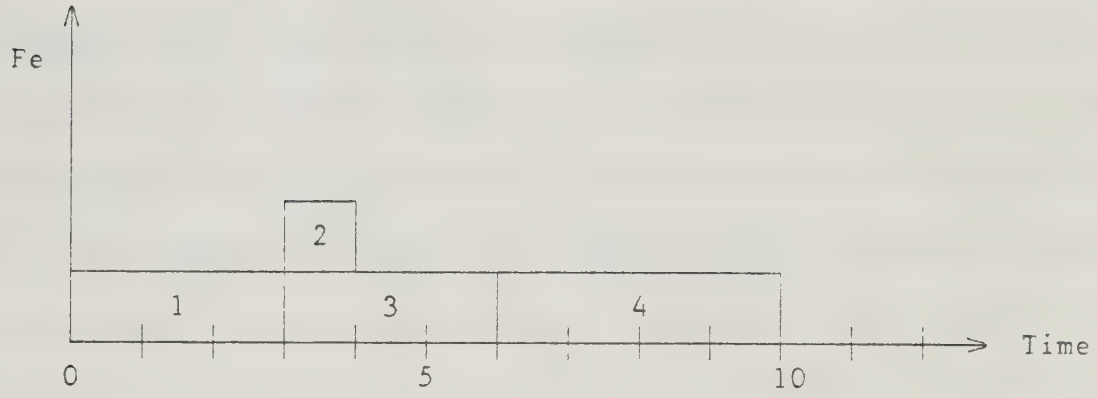
where l is the total number of levels. The bounds developed by Hu and Ramamoorthy are only valid for equal task execution times. Obviously, they are also applicable to graphs with unequal task times if tasks requiring T > 1 units of time are first expanded into T unit-time tasks. But this modification complicates the graph.

Fernandez and Bussell [FERN73] presented a new formulation which gives sharper lower and upper bounds. Let Fe be the execution profile of all the tasks according to the earliest completion times and Fl corresponding to latest completion times. For any interval $[\theta_1,\theta_2]$, $|Fe \cap Fl|$ is the set of tasks that cannot be processed earlier or later than this interval, or it is the total processing activity required within this interval. Note that repeated elements are not deleted from the set intersection. The lower bound on the minimum number of processors can now be expressed as

$$n = \left\lceil \max_{[\theta_1,\theta_2]} \left[\frac{1}{\theta_2-\theta_1} \; | F_e \cap F_l | \right] \right\rceil$$

where maximum is taken over all possible time intervals $[\theta_1,\theta_2]$. Figure 5.1 shows a graph where task execution times are given adjacent to the vertices. The critical path length for this graph is 10 time units. It is clear that at least two processors are needed to complete this set of partially ordered tasks in 10 time units.





(b)

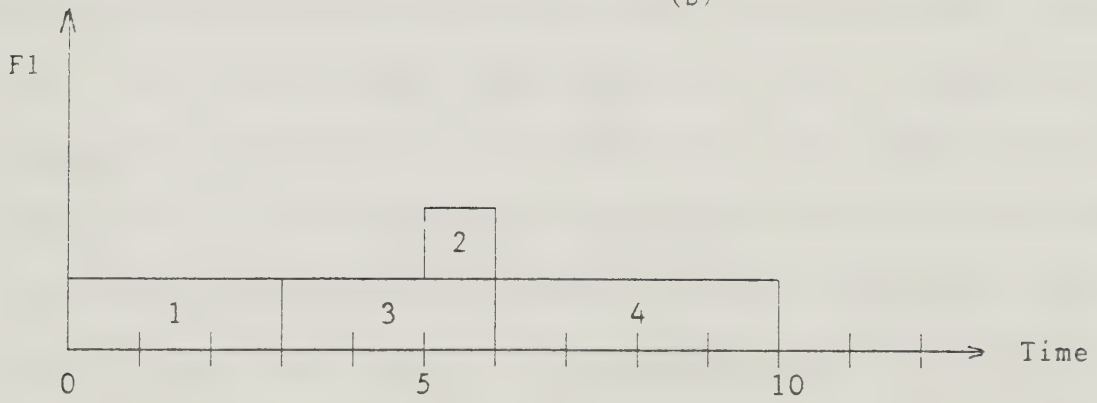


Figure 5.1 (a) Computation graph
(b) Fe and Fl

Yang [YANG76] proposed two new algorithms, based on Fernandez and Bussell's idea, for calculating the bounds on the number of processors. These algorithms use arithmetic operations, as opposed to slower intersection and union operations required by Fernandez and Bussell's method, to calculate the number of levels [YANG76] and precedence partitions (earliest and latest) of the graph. The time complexity of the resulting algorithm is $O(V^2)$, where V is the maximum number of levels in the graph.

In [FERN73] it is proved that this lower bound is sharper than the bounds of Chen and Epley, Hu, Kraska, and Ramamoorthy et. al. Moreover, this method takes a very balanced view of the graph since all the integer intervals are considered. Thus this method could be extended to suit our model. There are certain problems that need to be resolved before these ideas could be applied here. A critical path in our model will contain both task execution and inter-task communication cost. Recall that two tasks will incur inter-task communication cost if they are assigned to different processors but the cost of such communication is zero if these tasks reside on the same processor. Consider two tasks A and B which are located in the critical path and A is a predecessor of B; if A and B are assigned to the same processor then the critical path cost is reduced exactly by the communication cost from task A to B. So the critical path will be dependent on the task assignment. Thus it is inappropriate to define a critical

path in our model of the program.

Secondly, all the methods discussed so far calculate the lower bound on processors assuming all the tasks must be completed within the critical path length. As the previous discussion indicates, critical path length cannot be a basis for a new lower bound in our model. In addition, earliest and latest completion times also cannot be uniquely defined for there is no clear-cut way of accounting for communication costs. Thus we have two choices : either make some simplifications in our model or present another formulation.

One obvious simplification is to ignore the communication cost altogether and then find a lower bound on the minimum number of processors using one of the methods discussed earlier. But there are some pitfalls in this simplification. The number of processors required to execute the set of tasks given in Figure 5.1 is two if communication costs are ignored. Assume that communication cost between any pair of tasks is 3 units, then the finish time on two processors considering inter-task communication time is 13 units as shown in Figure 5.2. But these same set of tasks can be assigned to a processor and the finish time in that case would be 11 units. Thus high communication cost effectively reduces the parallelism in the graph and so neglecting communication cost can give too high a value of lower bound on the minimum number of processors.

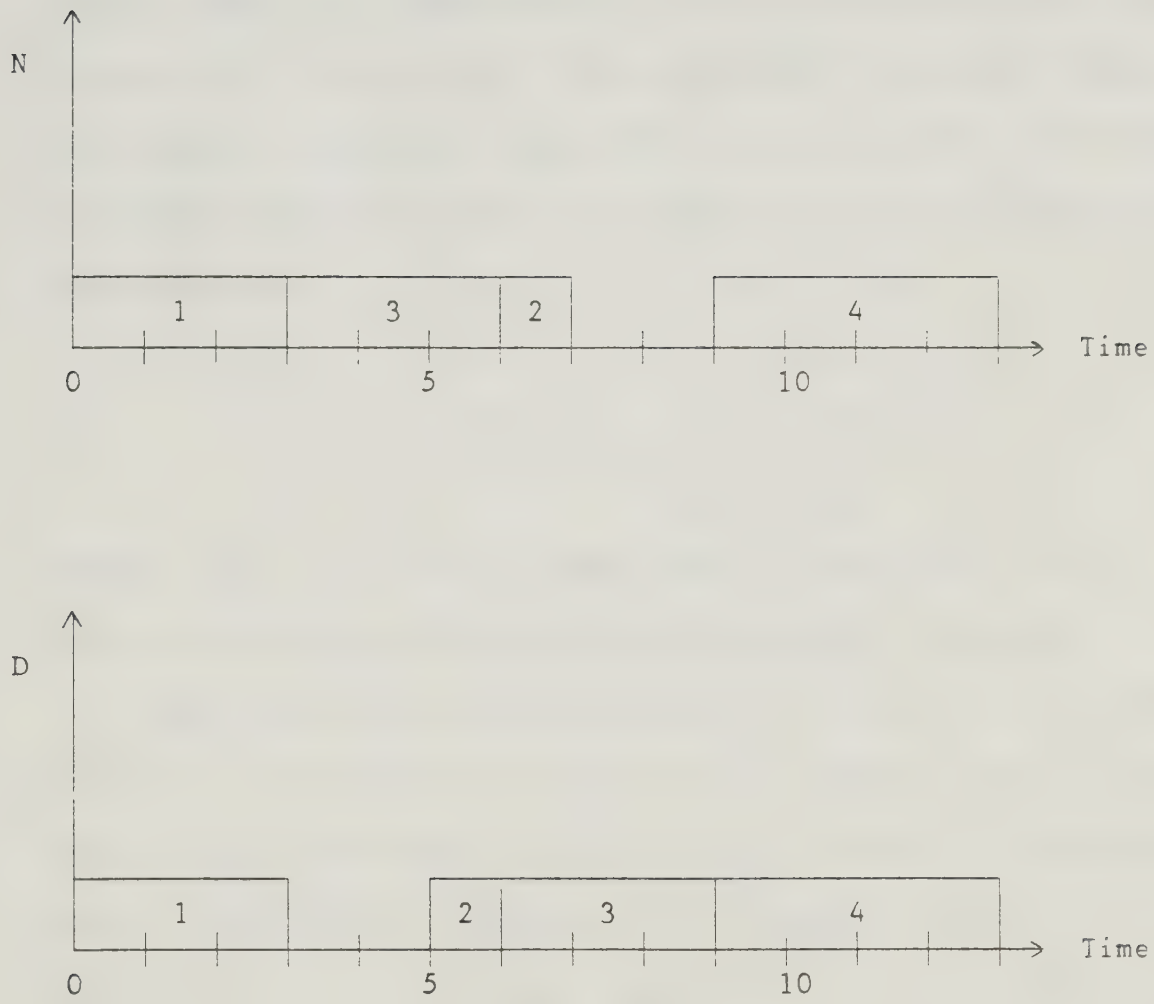


Figure 5.2 Normal and delayed activity plot of Figure 5.1(a) with inter-task communication cost of 3 units

5.2 A New Approach

In the last chapter we presented an algorithm that assigns a set of tasks on an unlimited number of processors. The basic property of this algorithm is that it exploits the parallelism inherent among tasks to minimize the finish time. We can use this algorithm to determine the lower bound.

Our approach is simple. We use the algorithm on the graph G and plot the activity of all the tasks and call it *normal activity plot* N. We also apply the same algorithm on the graph G and plot the activity of all the tasks and call it *delayed activity plot* D. Thus, the lower bound can be expressed as

$$n = \max \left\lceil \frac{1}{\theta_2 - \theta_1} \left\lceil |N \cap D| \right\rceil \right\rceil \quad \theta_1, \theta_2 \in t_{min}$$

where t_{min} is the finish time of the sink vertex. Note that the intersection contains repeated elements.

The normal and delayed activity plot of the graph of Figure 5.1 is shown in Figure 5.2. Notice that in the normal activity plot, task 4 cannot commence execution before time unit 9 because of the communication delay from task 3 to 4. It is clear that one processor should be sufficient to finish in minimum time.

The proposed lower bound calculation requires processing of $O(t_{min})$ intervals as there are $t_{min} * (t_{min}) / 2$

intervals. Moreover, the intersection operation required is slow in comparison to arithmetic operations. Fernandez et al. [FERN75] suggest ways to improve computational requirements for special cases (e.g. trees, independent tasks).

Observe that normal and delayed activity plot will indicate the maximum number of tasks simultaneously executable in any interval. Thus we can take the maximum over $[0,t_{min}]$ interval of the normal activity plot and call it N_{max}. Similarly D_{max} will be the maximum number of tasks active in any interval between $[0,t_{min}]$ in the delayed activity plot. Since we have the choice to schedule the tasks according to their normal or delayed activity plot, the upper bound on the minimum number of processors would be the smaller of N_{max} and D_{max}, or

$$n = \min [N_{max}, D_{max}]$$

It is straightforward to show that this upper bound can be calculated in a time complexity $O(t_{min})$.

Chapter 6

Facilitating Inter-task Communication in Distributed Systems

The result of the assignment algorithm is the set of tasks and processors that are assigned to these tasks. After the assignment is complete, the system executive transfers the tasks and their data (if available) to appropriate processors. Some tasks will have data values available to them before they are transferred to a processor. But some other tasks will require values computed by their predecessor tasks. As soon as the data values become available to a task, it can be executed on the processor assigned to it. Since distributed systems do not have a common shared virtual memory (as in multiprocessors), some sort of communication mechanism is required to enable a task to send values to other task(s). This type of communication is usually accomplished by transferring messages (containing data values) from the source (sender) to the destination (receiver) task.

In multiprocessors, this communication is accomplished by a common shared memory. The tasks read and write a set of variables in the common memory. The situation may arise where a set of tasks are attempting to read or write a memory location at the same time. This condition is referred to as *memory contention* and is believed to be a major problem as well as a bottleneck in multiprocessor systems. This entails use of synchronization primitives to

ensure correct order of read or write operations on memory locations.

For the same reason, distributed systems are advocated where communication takes place by the passing of messages between tasks. The task A intending to send a value to task B residing at processor Pj cannot directly write into the memory attached to Pj so that task B can read it. Instead, task A would send a message (containing values to be transferred) to task B. To facilitate this type of communication, the system executive would provide a set of standard communication primitives that could be used by tasks in lieu of expecting tasks to perform all the necessary processing involved in message transfer.

The part of the distributed system (hardware, software or a combination of both) that adds these communication primitives is of no immediate concern to us. We are interested in requirements that inter-task communication primitives must satisfy. That is, there are certain design goals that should be followed when designing the communication primitives.

In the following sections we discuss these design goals. Ideas presented are not restricted to any particular system structure or communication architecture but we strive to make it as general as possible.

6.1 The System Structure

Before we move on to inter-task communication mechanisms, it will be helpful to elaborate on the system structure.

A processing unit in this structure has the capabilities of a conventional micro-processor, but its functions and architecture are substantially different. It can perform local computation, initiate actions for fetching information from other nodes of the system, disseminate information, perform basic operating system functions and error detection and recovery. For the sake of exposition, the processors in the system are numbered from 0 to n-1 where n is the total number of processors in the system.

The primary memory of the system is distributed among the processing units. Each processing unit has direct access to the segment of memory located within it. The data from a non-local memory can be accesses by requesting the processor attached to that memory module. This request takes the form of a message transmitted across the the interconnection network.

6.2 Design Principles

Distributed systems can vary in size, capabilities, range of services offered, size of the processors, communication subsystems, and the like. But if the system were to execute a set of tasks correctly, they must be able

to communicate with each other regardless of the other characteristics of the system described above. That is, the system structure must be able to support the inter-task communication mechanism irrespective of other characteristics of the system.

The inter-task communication mechanism will take the form of a set of primitives like "send", "receive", "reply", "get", "put" etc. The tasks make use of these primitives to communicate with each other. Consider the case where task A has to send a message (containing some value) to task B; it will use the primitive "send" which contains the value and the address of the task (and other information e.g. checksum bits, header, trailer etc., which are of no concern to us). This immediately confronts us with the problem of deciding about the "address of the task". What should the address of the task contain : the address of the processor where task B resides or the address of the task itself or some hierarchical address (discussed later in the chapter). This may become an issue of some importance in the design of inter-task communication mechanisms.

Similarly, another design issue could be the scaleability of the distributed system itself : is the communication mechanism general enough to withstand expansion of the system?

These and other design issues are discussed in following paragraphs. Since we are neither designing a distributed system nor do we have any particular system in

mind, a general framework will be presented. The design objective presented are general enough to be applicable to any configuration or structure of the distributed system. The current designs do consider some of the objectives but a unified view of the inter-task communication problem is still lacking in the literature.

6.2.1 Independence from task assignment

6.2.1.1 Processor addressing

In its simplest form, inter-task communication can be carried out by processor to processor message transfers. When task A sends a message to task B residing on processor Pj, task A designates the recipient as processor Pj. It is implicit in this type of communication that processor Pj knows that this particular message is intended for task B. The processor Pj keeps a table of all the tasks running or waiting for processor time on it. Whenever a message arrives for Pj, it searches it's table to look for the task this message is intended for. Since a task never needs to find out the address of a sender, it would not make any difference whether the processors share a common address space or not.

Processor addressing has several advantages. Since the number of processors is usually less than the number of tasks running or waiting on these processors, some saving in the message length is possible (compared to the task

addressing scheme discussed later in this section). This could be attractive in light of the fact that until now communication network has been the slowest (in bytes transferred per instruction) part of the system. Moreover, routing of messages can be optimized with respect to total routing path length as the destination address is known beforehand.

Processor addressing is not without drawbacks. The task size (and consequently the number of tasks) and assignment of these tasks to processors would depend on the number of processors in the system. It is highly desirable that a software system should be able to execute on a range of hardware organizations with different degrees of distribution without extensive recompilation or reassignment. Consider a program that has been partitioned into tasks A, B, and C. Tasks A and B are assigned to a processor and C to a different processor. Since A and B reside on the same processor, they may be communicating via processor's local memory (as in centralized systems). Or they may be communicating using processor addressing. In case B is moved to another processor, it will amount to changing processor addresses in the messages from task A to B. The situation could be much worse if A and B share data also. Thus migration of tasks to different processors could involve changing the contents (destination address) of messages.

6.2.1.2 Task addressing

To alleviate this problem, a uniform communication mechanism is required. This amounts to saying that no distinction should be made in inter-task communication between two tasks residing on the same processor or different processors. This could be achieved if the sending task addresses messages directly to the task receiving it. The network interface unit (NIU) at the receiving processor (where the receiving task resides) is responsible for accepting the message.

The task addressing mechanism can be implemented in two different ways – destination and source task addressing [FRAN81]. The destination task addressing approach assumes that the messages are addressed to the receiver by the receiving task's name, whereas in the source task addressing approach the messages include the name of the source task (which implies the receiving task). An advantage [FRAN81] of destination addressing is that it facilitates compile-time checking for any violation of legal message exchanges since all the destination tasks are known. The source addressing mechanism advocates modular software since a task never needs to know the name of the destination task.

A major advantage [STOU79] of the task addressing approach is that tasks can be dynamically relocated. Carrying on with the previous example, no changes need to be made to tasks A and B even if they are moved to different processors. Another implication of task addressing is that

a program can be partitioned into an appropriate number of tasks so as to extract maximum parallelism and, then, these tasks can be assigned to available processors without bothering about relocation. Moreover, multi-destination messages are possible by allowing duplication of task names. Since multi-destination messages are possible, reliability can be increased by executing identical tasks on different processors through parallel redundancy.

Notice that task A sending a message to task B is not aware of the physical location of task B and vice versa. Thus either task cannot accidentally or maliciously overwrite a location in other task (assuming that there is no system-wide addressing scheme). This gives rise to some form of security in the system as the physical identity of a task is hidden from other tasks.

Task addressing is not the perfect answer for our problems. Since message passing is accomplished by naming the receiving task, it is implicit that there should be a system wide task-naming scheme. This may not be a serious problem if the system is localized in a physical or geographical location (say a building or a room). In that case this requirement can be enforced quite easily. But it also implies that every NIU be allowed to examine the messages whether or not they are intended for this particular processor. Also since a task can reside at any processor site, it is implicit that a particular message be transmitted to all the processors in the system. This will

be a big burden on the communication subsystem except, perhaps, for certain kind of interconnection structures like a star or a ring.

6.2.1.3 Hierarchical addressing

A simple solution would be to use a hierarchical addressing scheme. The n processors are divided into logical clusters of c processors each and there are n/c clusters. The processors in a cluster are in physical proximity and so communication between these processors is cheaper than inter-cluster communication. Each of these clusters is assigned an address. The task address would now consist of two parts – the cluster address and the address of the task. Let us assume that task A (residing in cluster x) feels the need to send a message to task B (residing in cluster z), then the first thing NIU (of the processor where task A resides) does is to find out the shortest (or cheapest) path to cluster z. The processor, in cluster z, that receives this message communicates it to all the processors in the cluster. Since task B is at one of the processors in cluster z, this message will eventually be accepted. Even though this modification involves some processing overhead at the sender and the receiver processors, it obviates excessive use of communication subsystem. Basically, this can be considered as a two part addressing scheme. It can be extended to three, four, ... part addressing schemes if some gain in system throughput is

realized.

A choice between task and processor addressing depends on some complex tradeoffs between efficiency on one hand and reliability, security, decentralization, etc. on the other hand. It is difficult to be definitive about either of the two choices. It all depends on the system objectives and kind of applications the system is intended for. But if we take a synergistic view of other advantages of the distributed systems (discussed in the following sections) then the choice lies in the task addressing approach.

In conclusion, a task addressing scheme is essential for allowing flexibility, graceful expansion, security, and fault tolerance of the system.

6.2.2 Extensibility

Except for special purpose, dedicated distributed systems designed with particular application in mind, extensibility will be an important design criteria. Extensibility refers to minimal change in inter-task communication mechanism even with the addition of more processors or functionality.

To see how extensibility (in the sense of expansion) comes into picture, consider the following situation. Processors can be added to a distributed system if the communication network is not saturated. The task addressing approach will ensure that no changes need to be made to inter-task communication mechanism because tasks are

addressed by names and not processor addresses. The processor addressing approach will require informing all the old processors about the addresses of new processors and vice versa. This will involve updating routing tables in all the processors.

Once the communication network is saturated, addition of more processors is not possible without restructuring the system or using a higher bandwidth network. The second alternative may not always be feasible because higher costs would be involved. Or the current technology may not permit it.

We again have some leeway in the second alternative. It should be possible to simplify the interconnection network (say from a fully connected network to a partially connected network). But some flexibility is lost in the process. If restructuring of the network is deemed impractical, then it would be necessary to use gateways to connect the new processors to the old network. Basically it divides all the processors into two logical groups : processors connected by the old network, and the set of new processors. In that case, it would be advantageous to use a hierarchical address instead of using task addressing. But this demands a change in the inter-task communication mechanism.

Extensibility seldom demands reconfigurability of the system elements – be it in the form of altering the interconnection network, or changing the functionality of

the processors with respect to inter-task communication mechanism, or just changing the degree of system-wide executive control. Whatever may be the case, the solution will inevitably gravitate towards flexibility offered by inter-task communication mechanism. It is essential that the inter-task communication mechanism be flexible enough to withstand all these changes. The preceding discussion suggests that a source task addressing may be the answer to these requirements. This type of addressing fosters modularity in software and reconfiguration by virtue of not requiring the identity of the destination task.

Thus the inter-task communication mechanism should be designed in such a way as to allow graceful expansion of the system (in number of processors, and functionality) with minimum of effort.

6.2.3 Generality

Any inter-task communication mechanism should be able to handle a wide range of data types *viz.* characters, integers, arrays and structured values. There are a number of reasons for this requirement.

Consider an inter-task communication mechanism that can only handle characters. If some structured value is to be sent to some other task in the system, it would require coding at the sender end, and decoding at the receiver end. Presumably, coding and decoding structured values will be time consuming and would, therefore, be totally

unacceptable. In the process, we would have also assumed that the processors are powerful enough to do all the necessary encoding and decoding without degrading the system performance. This is a tall order and may not hold for systems made up of microprocessors.

In addition to the time consuming coding and decoding process, transmission of the extra code bits will also be required, and thus becomes a burden on the communication network. Since the communication network itself is a bottleneck in the system, it is unwise to use it more than necessary.

Further, if a new structured value is to be added to the repertoire of already existing structured values, it would require informing all the processors in the system (in all 2*n programs – coding as well decoding – need to be changed).

The solution would be to support a set of orthogonal data types from which other data types could be easily constructed. As for the type of data structures that can qualify as being general in nature, an illuminating example would be a tree structure. Recently, researchers have tried to represent other data structures in tree-like structures [ROSE79].

Thus for the reasons of efficiency, inter-task communication should support a wide range of structured data types.

6.2.4 Debugging Facilities

If experience with serial programs is any indication, debugging distributed programs will be extremely difficult. We say this not only because of the nature of distributed programs, but the problems of timing, synchronization and monitoring concurrent execution play an important part also. In a serial program, it is easy to halt the execution of a program at a definite time instant or instruction. The state of the processor can be saved and restored easily. It is sufficient to know the state of the processor to detect a bug in the program. Whereas in a distributed system it is not possible to stop [LAMP78] a set of tasks running on different processors as accurate synchronization of time-of-day clocks at each processor is inconceivable. Thus it would be better if inter-task communication is monitored instead of stopping all the tasks and examining their history [SCHI81]. A good discussion of the problems of debugging distributed systems can be found in [SCHI81]. Since monitoring inter-task communication traffic is a major consideration for debugging tasks, it is essential that all the information a user needs to know be available or easily derivable from inter-task messages. A user will be interested in "sender" and "receiver" task names, contents of messages (including type and value of data), timing information, and the like. The type (of data) information is needed because a bit-string can interpreted as integers, reals, characters etc. It will be helpful to know what this

bit-string represents and thus type information is needed. These factors have to be taken into account when designing inter-task communication systems.

Chapter 7

Conclusions

Several issues related to the design of distributed systems have been discussed in this thesis. We pointed out that distributed systems offer advantages that are not attainable by other structures *viz* uniprocessors, multiprocessors, or networks. But before these benefits can be realized, substantial research needs to be done in resolving certain issues that are unique to distributed systems. These areas include :

1. partitioning of programs into tasks,
2. assignment of these tasks to available processors, and
3. designing inter-task communication mechanism.

It should be pointed out that the above list is by no means exhaustive; it serves to identify just the basic problems in distributed systems. The main theme of this thesis is centered around the problems stated in the preceding list.

As observed by Enslow [ENSL78], and more recently by Franta et al [FRAN81], program partitioning techniques are still rudimentary in nature. Our study indicates that partitioning a program merely on the basis of parallelism among different statements will not be fruitful. This tends to neglect other types of parallelism existing in the program – for example, within arithmetic expressions, within iterative statements, and between procedures. A method is needed that can extract parallelism at a much more microscopic level. We suggested that a good candidate for

tackling this problem is the data flow approach. The data flow approach is simple, and yet powerful enough to express the parallelism in programs.

There are two different ways to obtain the data flow representation of an algorithm : either use a data flow language such as VAL or Id, or use a conventional high-level procedural language (e.g. ALGOLW, or ALGOL 68). Data flow languages are still in the state of development and therefore their use is unpractical at this point of time. Recently, some progress has been made [ALLA79] in translating a program written in a procedural language into a data flow language program. Our approach would be to write the program in a procedural language, translate it into a data flow representation, and then partition it into tasks. We also presented arguments for following this approach.

The assignment of tasks is another major issue. The problem of optimal assignment is NP-complete and thus we favor designing efficient heuristics. To the best of our knowledge, no attention has been made to design heuristic algorithms that take precedence relationships between tasks into account while performing the assignment. We, then, suggest a new heuristic algorithm that considers precedence relationships between tasks. The complexity of the algorithm is shown to be linear in the number of vertices and edges ($O(m + |E|)$) where vertices represent tasks and edges denote inter-task communication. An application of

this algorithm to find the bounds on the number of processor is also suggested.

An important part of the success of any distributed system would be the characteristics of the inter-task communication mechanism(s). Distributed systems are still at the research stage and there is no clear-cut framework for designing communication mechanism. In the last chapter, we have tried to present some of the issues involved in such a design. These issues are by no means complete *per se*; they are meant to be as widely applicable as possible. Of course, certain special purpose distributed systems would impose different requirements of communication mechanisms. But we do not concern ourselves with these special systems and their requirements as we do not want to restrict our horizon. In our opinion, these are the minimum set of requirements that must be considered when designing a distributed system communication mechanism.

Some further areas for investigation are now suggested.

7.1 Areas for future investigations

There are a number of problems in the distributed systems area for which satisfactory answers are yet to be found. Some of these problems are common to centralized as well as distributed systems, but the solution found in the case of a centralized system cannot (generally) be extended to distributed systems. A case in point would be

synchronization methods. A centralized system can use synchronization primitives like semaphores, conditional critical regions, path expressions etc. But these primitives are not applicable in distributed systems as the communication network introduces its own problems. Moreover, there is no shared common memory among processors. Thus distributed systems introduce a whole spectrum of challenging problems that currently defy solution (for a good discussion of problems involved in designing and implementing distributed systems, the reader may refer to [ECKH78]).

7.1.1 Inter-task communication facilities

A major factor in the performance of distributed systems is that inter-task messages should be transmitted efficiently. There are two types of delays in message transmission : message processing and transmission delay across the network. It should be possible to implement message processing in either hardware (exemplified by HXDP [FRAN81]) or software. But experience with the HYDRA operating system running on C.mmp [SWAN77] indicates that this function should be implemented in hardware rather than in software. Since hardware cost are rapidly declining with the advent of VLSI, it will not add substantially to the total system cost. However, some part of message processing still needs to be implemented in software.

We immediately observe that several interesting possibilities (in distributing the processing between hardware and software) arise. It is not possible to suggest a straightforward answer to this problem. But we know that hardware could be made reliable, by duplicating functional units, and can operate faster if the technology permits, whereas software offers ease of modification. Thus solving this problem depends on some complex tradeoffs between speed and reliability on one hand, and ease of modification on the other. It would be useful if some demarcation line could be drawn between the areas inherently more suitable for implementation of message processing in hardware and in software.

More baffling is the problem of coming up with an orthogonal set of communication primitives. By this we mean a set of communication primitives that are functionally complete and the capabilities offered by a member of the set is not a subset of any other member. Researchers designing HXDP (Honeywell Experimental Distributed Processor) [FRAN81] decided on three primitives : *send, receive, and wait*. This set of primitives is informally proven to be complete and the minimum possible for their application requirements. But it is quite possible that the set may not be complete for some other applications. Pipeline processing, for example, is not possible with just these three primitives. Research is required in resolving this issue.

There is a lack of theoretical foundations for communication mechanisms and protocols. Saltzer [SALT78] pointed out that "semantics for requesting operations, and reporting results and failures are needed". Theory is important so as not to leave any ambiguity and uncertainity in the solution of the problem. The solution can be verified if we have the necessary theoretical model and tools to realize this model. The reader is referred to the paper by Saltzer [SALT78] for a good overview of research problems in distributed systems.

7.1.2 Partitioning of programs

Identifying parallelism in a program is only the first step to make it suitable for execution on a distributed system. More important is the problem of data distribution among tasks (or processors) in the system. Experiments performed on Cm* indicate [DEMI82] that location of the data becomes a key issue if the speed of the tasks depends on the location of the data the task accesses. This can be attributed to the fact that access to the local data is usually much faster than remote data access. The time for remote data access in Cm*, for example, is atleast 2.5 times greater than local data access [DEMI82]. Thus algorithms need to be developed to distribute data among tasks (or processors) depending on the frequency of access by tasks.

Another interesting research problem is to determine the type of interconnection structure(s) most suitable for a

particular application. Sometimes the structure of the algorithm itself suggests the interconnection structure that should be used. But the number of problems that can be classified in this manner are limited. A problem may exhibit different structures (e.g. tree, pipeline) as the execution progresses. The problem, then, becomes one of finding the most suitable structure for any given application. This information would be useful in the partitioning process.

7.1.3 Assignment of tasks

We presented a heuristic algorithm for static assignment of tasks to processors. In other words, tasks cannot be moved to other processors if need arises, or dynamic reassignment is not possible. A criteria is needed for dynamic reassignment. But it is not evident what criteria should be used for this kind of load balancing [ECKH78]. These criteria could include, e.g., initial assignment, data accessing patterns, structure of the problem, processor load, inter-task communication time, ease of redirecting messages, etc. The solution to the general reassignment problem (we surmise) would be extremely hard; our approach would be to solve the problem for specific problem structures and interconnection networks.

There is no dearth of problems in the distributed systems area as it is still in its infancy. Substantial research needs to be done before all the benefits promised

by distributed systems can be realized.

8. References

ACKE82 Ackerman, W.B., "Data flow languages," *Computer*, vol. 15, no. 2, pp. 15-25, February 1982

ADAM74 Adam, T.L., K.M. Chandy and J.R. Dickson, "A comparison of list schedules for parallel processing systems," *Comm. Ass. Comput. Mach.*, vol. 17, no. 12, pp. 685-690, December 1974

ALLA79 Allan, S.J. and A.E. Oldehoeft, "A flow analysis procedure for the translation of high level languages to a data flow language," *Proc. 1979 Int. Conf. Parallel processing*, pp. 26-34, August 1979

ARNO82 Arnold, R.G., R.O. Berg and J.W. Thomas, "A modular approach to real-time supersystems," *IEEE Trans. Comput.*, vol. C-31, no. 5, pp. 385-398, May 1982

AROR80 Arora, R.K. and S.P. Rana, "Heuristic algorithms for process assignment in distributed computing systems," *Information Processing Lett.*, vol. 11, no. 4-5, pp. 199-203, December 1980

BARS68 Barskiy, A.B., "Minimizing the number of computing devices needed to realize a computational process within a specified time," *Eng. Cybern.*, no. 6,

pp. 59-63, June 1968

BERN66 Bernstein, A.J., "Analysis of programs for parallel processing," *IEEE Trans. Comput.*, vol. 15, no. 5, pp. 757-763, October 1966

BUSS74 Bussell, B., E. Fernandez and O. Levy, "Optimal scheduling for homogeneous multiprocessors," *Information Processing 74*, North-Holland Publishing Co., pp. 286-290, 1974

CHEN68 Chen, Y.E. and D.L. Epley, "Bounds on memory requirements of multiprocessor systems," in *Proc. 6th Annual Allerton Conf. Circuit and Syst. Theory*, pp. 523-531, 1968

CHRI81 Christopher, T.W. et al, "SALAD : A distributed compiler for distributed systems," *Proc. 1981 Int. Conf. Parallel Processing*, pp. 50-57, August 1981

CHUW69 Chu, W.W., "Optimal file allocation in multiple computing system," *IEEE Trans. Comput.*, vol. C-18, no. 10, pp. 885-889, October 1969

CHUW78 Chu W.W., D.Lee and B. Iffla, "A distributed processing system for Naval Data Communication Network," *NCC 1978*, pp. 783-793, 1978

CHUW80 Chu, W.W., L.J. Holloway, M.T. Lan and K. Efe, "Task allocation in distributed data processing," *Computer*, vol. 13, no. 11, pp. 57-69, November 1980

DEMI82 Deminet, J., "Experience with multiprocessor algorithms," *IEEE Trans. Comput.*, vol. C-31, no. 4, pp. 278-287, April 1982

DENN80 Dennis, J.B., "Data flow supercomputers," *Computer*, vol. 13, no. 11, pp. 48-56, November 1980

ECKE77 Ecker, K., "On task scheduling in an multiprocessor environment," in *Dig. COMPCON Fall 77*, pp. 297-298, September 1977

ECKH78 Eckhouse, R.H., Jr. and J.A. Stankovic, "Issues in distributed processing," *IEEE Computer*, vol. 11. no. 1, pp. 22-27, January 1978

EFEK82 Efe K., "Heuristic models of task assignment scheduling in distributed systems," *Computer*, vol. 15, no. 6, pp. 50-56, June 1982

ELDE79 El-Dessouki, O., W. Huen and M. Evens, "Towards a partitioning compiler for a distributed computing system," *Proc. First Int. Conf. on Distributed Computing*, pp. 296-304, October 1979

ENSL78 Enslow, P.H., Jr., "What is a Distributed Data Processing system," *IEEE Computer*, vol. 11, no. 1, pp. 13-21, January 1978

FERN73 Fernandez, E. and B. Bussell, "Bounds on the number of processors and time for multiprocessor optimal schedules," *IEEE Trans. Comput.*, vol. C-22, no. 8, pp. 745-751, August 1973

FERN75 Fernandez, E. and T. Lang, "Computation of lower bounds for multiprocessor schedules," *IBM J. Res. Develop.* , pp. 435-444, September 1975

FISH67 Fisher, D.A., "Program analysis for multiprocessing," *Burroughs Corporation*, May 1967

FLYN66 Flynn, M.J., "Very high-speed computing systems," *Proc. IEEE*, vol. 54, no. 12, pp. 1901-1909, December 1966

FRAN81 Franta, W.R. et al "Real-Time Distributed Computer Systems," in *Advances in Computers*, vol. 20, pp. 40-82, Academic Press, 1981

GYLY76 Gylys, V.B. and J.A. Edwards, "Optimal partitioning of workload for distributed systems," in *Dig. COMPCON Fall 1976*, pp. 353-357, 1976

HANS78 Hansen, B., "Distributed Processes : A concurrent programming concept," *Commun. ACM*, vol. 21, no. 11, pp. 934-941, November 1978

HOAR78 Hoare, C.A.R., "Communicating Sequential Processes," *Commun. ACM*, vol. 21, no. 8, pp. 666-677, August 1978

HORO76 Horowitz, E. and S. Sahni, *Fundamentals of Data Structures* , Computer Science Press, Rockville, Maryland, 1976

HORO78 Horowitz, E. and S. Sahni, *Fundamentals of Computer Algorithms* , Computer Science Press, Rockville, Maryland, 1978

HUEN77 Huen, W., O. El-Dessouki, E. Huske and M. Evens, "A pipelined DYNAMO compiler," *Proc. 1977 Int. Conf. Parallel Processing*, pp. 57-66, August 1976

HUTC61 Hu, T.C., "Parallel sequencing and assembly line problems," *Oper. Res.*, vol. 9, pp. 841-848, November 1961

IGNI76 Ignizio, J.P., *Goal Programming and Extensions*, D.C. Heath and Co., Indianapolis, Indiana, 1976

IGNI82 Ignizio, J.P., D.F. Palmer and C.M. Murphy, "A multicriteria approach to supersystem architecture definition," *IEEE Trans. Comput.*, vol. C-31, no. 5, pp. 410-418, May 1982

JENN77 Jenny, C.J., "Process partitioning in distributed systems," in *Dig. Papers NTC 1977*, pp. 1-10, 1977

JENS76 Jensen, E. and W. Boebert, "Partitioning and assignment of distributed processing software," in *Dig. Papers COMPCON Fall 1976*, pp. 348-352, September 1976

JENS77 Jensen, J.E., "A fixed variable scheduling model for multiprocessors," *Proc. 1977 Int. Conf. Parallel Processing*, pp. 108-117, August 1977

JENS78 Jensen, E.D. "The Honeywell Experimental Distributed Processor—an overview," *IEEE Computer*, vol. 11, no. 1, pp. 28-38, January 1978

KOHL75 Kohler, W.H., "A preliminary evaluation of the critical path method for scheduling tasks on multiprocessors systems," *IEEE Trans. Comput.*, vol. C-24, no. 12, pp. 1235-1238, December 1975

KOHL76 Kohler, W.H. and K. Steiglitz, *Computer and Job-Shop*

Scheduling Theory, E.G. Coffman, Jr. Ed., John Wiley, New York, 1976

KRAS72 Kraska, P.W., "Parallelism exploitation and scheduling," Dept. Computer Science, Univ. Illinois, Urbana, Report UIUCDCS-R-62-518, June 1972

KUCK76 Kuck, D.J., "Parallel processing of ordinary programs," in *Advances in Computers*, vol. 15, pp. 119-179, Academic Press, 1976

KUCK79 Kuck, D.J. and D.A. Padua, "High-speed multiprocessors and their compilers," *Proc. 1979 Int. Conf. Parallel Processing*, pp. 5-16, August 1979

LAMP78 Lamport, L. "Time, clocks, and the ordering of events in a distributed system," *Comm. ACM*, vol. 21, pp. 558-565, July 1978

LAND82 Landweber, L.H., "A software-partitionable multicomputer : a testbed for research in distributed computing," Dept. of Computer Science, Univ. Wisconsin, Madison, 1981

LEER80 Lee, R.B., "Empirical results on the speed, efficiency, redundancy and quality of parallel

computations," *Proc. 1980 Int. Conf. Parallel Processing*, pp. 91-100, March 1980

LOVI81 Lo, V. and J.W.S. Liu, "Task assignment in distributed multiprocessor systems," *Proc. 1981 Int. Conf. Parallel Processing*, pp. 358-360, August 1981

MCCO72 McCormick, W.T., P.J.Schweitzer and T.W. White "Problem decomposition and data recognition by clustering technique," *Oper. Res.*, vol. 20, no. 5, pp. 993-1009, 1972

MAPR82 Ma, P.R., E.Y.S. Lee and M. Tsuchiya, "A task allocation model for distributed computing systems," *IEEE Trans. Comput.*,vol. C-31, no. 1, pp. 41-47, January 1982

NETT76 Nett, E., "On further applications of the Hu algorithm to scheduling problems," *Proc. 1976 Int. Conf. Parallel Processing*, pp. 317-325, August 1976

NETT77 Nett, E., "On scheduling algorithms for n-free task dependency structures," *Proc. 1977 Int. Conf. Parallel Processing*, pp. 100-107, August 1977

PRIC81 Price, C.C., "The assignment of computational tasks among processors in a distributed system," *NCC 1981*,

pp. 292-296, 1981

RAMA72 Ramamoorthy, C.V., K.M. Chandy and M.J. Gonzalez, "Optimal scheduling strategies in multiprocessor system," *IEEE Trans. Comput.*, vol. C-21, no. 2, pp. 137-146, February 1972

RAMA76 Ramamoorthy, C.V. and W.H. Leung, "A scheme for parallel execution of sequential programs," *Proc. 1976 Int. Conf. Parallel Processing*, pp. 312-316, August 1976,

RAOG79 Rao G.S., H.S. Stone and T.C. Hu, "Assignment of tasks in a distributed processor system with limited memory," *IEEE Trans. Comput.*, vol. C-28, no. 4, pp. 354-361, May 1979

ROSE79 Rosenberg, A.L., "Encoding data structures in trees," *JACM*, vol. 26, no. 4, pp. 668-689, October 1979

SALT78 Saltzer, J.H., "Research problems of decentralized systems with largely autonomous nodes," *ACM Oper. Syst. Rev.*, vol. 12, no. 1, pp. 43-52, January 1978

SCHI81 Schiffenbauer, R.D., *Interactive Debugging in a Distributed Computational Environment*, Laboratory

for Computer Science, MIT, TR-264, September 1981

STON76 Stone, H.S., "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Trans. Software Eng.*, vol. SE-3, no. 1, pp. 85-93, January 1977

STOU79 Stoustrup, B., "An inter-module communication system for a distributed system," *Proc. First Int. Conf. on Distributed Computing*, pp. 412-418, October 1979

SWAN77 Swan, R.J., S.H. Fuller and D.P. Siewiorek, "Cm*—A modular, multi-microprocessor," *NCC 77*, pp. 637-644, 1977

TESL68 Tesler, L.G. and H.J. Enea, "A language design for concurrent processes," *AFIPS Conf. Proc.*, vol. 32, pp. 403-408, 1968

ULLM73 Ullman, J.D., "Polynomial complete scheduling problems," *ACM Oper. Syst. Rev.*, vol. 7, pp. 96-101, October 1973

WANG79 Wang, P.S. and M.T. Liu, "Parallel processing of high-level language programs," *Proc. 1979 Int. Conf. Parallel Processing*, pp. 17-25, August 1979

WUSB80 Wu S.B. and M.T. Liu, "A partition algorithm for

parallel and distributed processing," *Proc. of the 1980 International Conference on Parallel Processing*, pp. 254-255, March 1980

YANG76 Yang, C.C., "Fast algorithms for bounding the performance of multiprocessor systems," *Proc. 1976 Int. Conf. Parallel Processing*, pp. 73-82, August 1976

Index